Quartus II Version 6.1 Handbook

Volume 1: Design & Synthesis

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

QII5V1-6.1

Contents

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

Chapter 4. Quartus II Support of HardCopy Series Devices

Chapter 5. Engineering Change Management

Section II. Design Guidelines

Chapter 6. Design Recommendations for Altera Devices

Chapter 7. Recommended HDL Coding Styles

Section III. Synthesis

Chapter 8. Quartus II Integrated Synthesis

Chapter 9. Synplicity Synplify & Synplify Pro Support

Chapter 10. Mentor Graphics LeonardoSpectrum Support

Chapter 11. Mentor Graphics Precision RTL Synthesis Support

Chapter 12. Synopsys Design Compiler FPGA Support

Chapter 13. Analyzing Designs with Quartus II Netlist Viewers

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 1*, 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: *November 2006*
Part number: *QII51016-6.1.0*

Chapter 2. Quartus II Incremental Compilation for Hierarchical & Team-Based Design
Revised: *November 2006*
Part number: *QII51015-6.1.0*

Chapter 3. Quartus II Design Flow for MAX+PLUS II Users
Revised: *November 2006*
Part number: *QII51002-6.1.0*

Chapter 4. Quartus II Support of HardCopy Series Devices
Revised: *November 2006*
Part number: *QII51004-6.1.0*

Chapter 5. Engineering Change Management
Revised: *November 2006*
Part number: *QII51005-6.1.0*

Chapter 6. Design Recommendations for Altera Devices
Revised: *November 2006*
Part number: *QII51006-6.1.0*

Chapter 7. Recommended HDL Coding Styles
Revised: *November 2006*
Part number: *QII51007-6.1.0*

Chapter 8. Quartus II Integrated Synthesis
Revised: *November 2006*
Part number: *QII51008-6.1.0*

Chapter 9. Synplicity Synplify & Synplify Pro Support
Revised: *November 2006*
Part number: *QII51009-6.1.0*

Chapter 10. Mentor Graphics LeonardoSpectrum Support
Revised: *November 2006*
Part number: *QII51010-6.1.0*

Chapter 11. Mentor Graphics Precision RTL Synthesis Support
Revised: *November 2006*
Part number: *QII51011-6.1.0*

Chapter 12. Synopsys Design Compiler FPGA Support
Revised: *November 2006*
Part number: *QII51014-6.1.0*

Chapter 13. Analyzing Designs with Quartus II Netlist Viewers
Revised: *November 2006*
Part number: *QII51013-6.1.0*

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 6.1.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 *(1)* (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	www.altera.com	www.altera.com
Altera literature services	literature@altera.com *(1)*	literature@altera.com *(1)*
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.altera.com

Note to table:
(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 6.1 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

This document uses the typographic conventions shown below.

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, filenames, filename 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.
✓, —, N/A	Used in table cells to indicate the following: ✓ indicates a "Yes" or "Applicable" statement; — indicates a "No" or "Not Supported" statement; N/A indicates that the table cell entry is not applicable to the item of interest.
■ ● •	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.

Visual Cue	Meaning
	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.

Section I. Design Flows

The Altera® Quartus® II, version 6.1 design software provides a complete multi-platform design environment that easily adapts to your specific design needs. The Quartus II software also allows you to use the Quartus II graphical user interface, EDA tool interface, or command-line interface for each phase of the design flow. This section explains the Quartus II, version 6.1 software options that are available for each of these flows.

This section includes the following chapters:

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

1. Design Planning with the Quartus II Software

QII51016-6.1.0

Introduction

Due to the significant increase in FPGA device densities over the last few years, designs are often partitioned into different blocks with multiple designers. The inherent flexibility of advanced FPGAs means that the pin layout, power consumption, and timing performance for each design block are all dependent on the final design implementation. The system architect must resolve these design issues during the integration phase, often leading to problems that affect the overall time to market and thereby increase cost. Many potential problems can be solved earlier in the design cycle by selecting the optimal device and programming method, properly planning I/O pin locations, estimating power consumption, performing good design partitioning, and obtaining early timing estimates.

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

This chapter provides an introduction to various design and planning features in the Quartus® II software. For a general overview of the Quartus II design flow and features, refer to the *Introduction to Quartus II Manual*. For more details about specific Quartus II features and methodologies, this chapter provides references to other appropriate chapters in the *Quartus II Handbook*.

Device & Programming/ Configuration Method Selection

The first stage in design planning is choosing the best device for your application and determining how you want to program or configure the device in your system. These factors affect the rest of your design cycle, including board specification and layout. Most of this planning is performed outside of the Quartus II software, but this section provides a few suggestions to aid in the planning process.

Device Selection

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

Determining the required device density can be a challenging part of the design planning process. Devices with more logic resources and higher I/O counts can implement larger and potentially more complex designs, but generally have a higher cost. Select a device that meets your design needs with some safety margin, in case you want to add more logic later in the design cycle. Consider needs for specific types of dedicated logic blocks, such as memory blocks of different sizes, or digital signal processing (DSP) blocks to implement certain arithmetic functions.

If you have prior designs targeting Altera devices, you can use their resource utilization as an estimate for your new design. Compile existing designs in the Quartus II software with an "Auto" device setting to review the resource utilization and find out which device density fits the design. Note that coding style, device architecture, and the optimization options used in the Quartus II software can significantly affect a design's resource utilization.

To obtain resource utilization estimates for certain configurations of Altera's intellectual property (IP) designs, refer to the User Guides for Altera Megafunctions and IP MegaCores on the **IP Megafunctions** page in the Literature section on the Altera website. You can use these numbers to help estimate the resource utilization of your design.

Device Migration Planning

Determine if you want the option of migrating your design to another device density to allow flexibility when the design nears completion, or migrating to a HardCopy structured ASIC device when the design reaches volume production. If so, you should specify these migration options in the Quartus II software at the beginning of your design cycle. Specify the target migration devices in the **Migration compatibility** section of the **Device** page in the **Settings** dialog box.

Selecting a migration device impacts pin placement and sometimes restricts logic utilization to ensure that your design is compatible with the selected device(s). Adding migration devices later in the design cycle is possible, but requires extra effort to check pin assignments, and may require design changes to fit into the new target device. It is much easier to consider these issues early in the design cycle than at the end, when the design is near completion and ready for migration.

Programming/Configuration Method Selection

Choosing your programming or configuration method up-front allows system and board designers to determine what companion devices, if any, are needed for your system. Your board layout also depends on the type of programming or configuration method you plan to use for programmable devices. Many programming options use a JTAG interface to connect to the devices, so your design may require a JTAG chain be set up on the board.

The device family handbooks describe the configuration options available for a given device family. For more details about configuration options, refer to the *Configuration Handbook*. Programming and configuration of Altera devices includes the following options:

- Using enhanced configuration devices—These devices combine industry-standard flash memory with a feature-rich configuration controller, including device features such as concurrent and dynamic configuration, data compression, clock division, and an external flash memory interface. You can also implement remote and local system updates with enhanced configuration devices.
- Using Flash memory devices with a memory controller, such as an Altera MAX® device—The flash memory controller can interface with a PC or microprocessor to receive configuration data via a parallel port.
- Using Quartus II Serial Flash Loader (SFL)—This scheme allows you to configure the FPGA and program serial configuration devices using the same JTAG interface.

- Using Quartus II Parallel Flash Loader (PFL)—This solution quickly retrieves data from a JTAG interface and generates data formatted for the receiving target flash device, significantly reducing the flash device programming time. If your system already contains a common flash interface (CFI) flash memory, you can utilize it for the FPGA configuration storage as well, because the PFL feature supports many common industry-standard flash devices. If you choose this method, you should check the list of supported flash devices early in your system design cycle and plan accordingly.

Early Planning Tools

You can use the Quartus II early planning tools to provide information to PCB board and system designers. Providing FPGA device information early in the design process enables earlier planning for power and board design requirements. You can perform early power estimation, as well as early pin planning and analysis, before you have created any source code, or when you have a preliminary version of the design, and then perform the most accurate analysis when the design is complete.

Early Power Estimation

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

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

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

If you have an existing design or a partially-completed design, the power estimator file generated by the Quartus II software can provide input to the spreadsheet for your current design (refer to "Early Power Estimator File" on page 1–5).

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

The results of the Power Estimator spreadsheet and Power Analyzer are only an estimation of power, not a specification. The estimation helps establish a guide for the design's power budget.

The PowerPlay Early Power Estimator spreadsheets for each supported device family are available in the Devices section under Support on the Altera website: **http://www.altera.com/support/devices/estimator/pow-powerplay.html**

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

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

Early Power Estimator File

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

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

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

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

Early Pin Planning & I/O Analysis

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

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

The Pin Planner enables easy I/O pin assignment planning, assignment, and validation. Use the Pin Planner Package view to make pin location and other assignments using a device package view instead of pin numbers. The Pads view displays I/O pads in order around the silicon die to help you follow pad distance and pin placement guidelines. With the Pin Planner, you can identify I/O banks, VREF groups, and differential pin pairings to help you through the I/O planning process. 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. You have the option to use a Microsoft Excel spreadsheet to start the process if you use a spreadsheet in your design flow, and you can export a Comma-Separated Value (**.csv**) file for use in a spreadsheet when all pins are assigned.

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

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

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

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

Early in the design process, before the source code has been created, the system architect typically has information about the I/O interfaces and IP cores that will be used in the design. Use this information with the **Create/Import Megafunction** feature in the Pin Planner to specify details about the design I/O interfaces.

The Pin Planner interfaces with the MegaWizard® Plug-In Manager, allowing you to create or import custom megafunctions and IP cores. Configure the way in which they will be connected to each other by specifying matching node names for selected ports in the **Set up Top-Level Design File** dialog box. Make any other I/O-related assignments for these interfaces or other design I/O pins in the Pin Planner.

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

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

For more information about Quartus II Revisions and Settings files, refer to the *Quartus II Project Management* chapter in volume 2 of the *Quartus II Handbook*.

Design & Compilation Flows

You should determine the design and compilation flow you will be using for your design. The following subsections describe the flat compilation flow, where the design hierarchy is flattened without design partitions, as well as incremental compilation flows that use design partitions in top-down, bottom-up, or mixed design methodologies. Incremental compilation flows offer several advantages but require more design planning to ensure good quality of results.

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

Flat Compilation Flow with No Design Partitions

In this compilation flow in the Quartus II software, the entire design is compiled together in a "flat" netlist. This flow is used if you do not create any design partitions. Your source code can have hierarchy, but the design is flattened during compilation and all of the design source code is synthesized and fit in the target device whenever the design is recompiled after any change in the design. This behavior helps obtain optimal quality of results. 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 debug tools incrementally, such as the SignalTap® II logic analyzer, but you do not specify any design partitions to preserve design hierarchy during compilation.

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

The standard Quartus II compilation flow consists of the following essential modules:

- Analysis & Synthesis—performs logic synthesis to minimize the design logic and performs technology mapping to implement the design logic using device resources, such as logic elements. This stage also generates the project database that integrates the design files, including netlists from third-party synthesis tools. When you are using EDIF or VQM netlists created by third-party synthesis tools, the Analysis & Synthesis stage performs logic synthesis and technology mapping only for black boxes and Altera megafunctions.

- Fitter—places and routes the logic of a design into a device.
- Assembler—converts the Fitter's device, logic, and pin assignments into programming files for the device.
- Timing Analyzer—analyzes and validates the timing performance of all logic in a design.

Incremental Compilation with Design Partitions

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

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

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

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

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

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

In this flow, after you partition the design, the software performs logic synthesis and technology mapping for each partition individually with Analysis & Synthesis. Analysis & Synthesis reads the project assignments to determine the partition boundaries. If any part of the design changes, Analysis & Synthesis processes the changed partitions and keeps the existing netlist for the unchanged partitions.

If you use a third-party synthesis tool, you should create separate VQM or EDIF netlists for each design partition in your synthesis tool. These netlists are then considered the "source file" for incremental compilation. After completion of Analysis & Synthesis, each partition has one post-synthesis netlist.

The Quartus II Partition Merge step creates a complete netlist that consists of post-synthesis, post-fitting netlists, or both, or netlists imported from lower-level projects, depending on the netlist type you specify for each partition. The Fitter then processes the merged netlist, preserving the placement or placement and routing of unchanged partitions, and refitting only those partitions that have changed.

The remainder of this section provides planning guidelines for incremental compilation, and describes the different types of incremental compilation flows so you can choose the flow that meets your design requirements.

Planning for Incremental Compilation

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

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

It is important to plan and balance resource utilization. When performing incremental compilation, the software synthesizes each partition separately, with no data about the resources used in other partitions.

Therefore, device resources can be overused in the individual partitions during synthesis, and the design may not fit in the target device when the partitions are merged.

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

You should create a design floorplan to avoid conflicts between design partitions, and to ensure that each partition is placed well relative to other partitions. Creating location assignments for each partition ensures that no conflicts occur for locations between different partitions. In addition, a design floorplan helps to avoid a situation in which the Fitter is directed to place or replace a portion of the design in an area of the device where most resources have already been claimed. This situation leads to increased compilation time and reduced quality of results.

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

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

For details about using the incremental compilation flows in the Quartus II software, as well as important guidelines for creating design partitions and a design floorplan, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

For details about creating placement assignments in the design floorplan, refer to the *Timing Closure Floorplan* and *Design Analysis and Engineering Change Management with Chip Planner* chapters in volume 2 of the *Quartus II Handbook*.

Top-Down Incremental Compilation Flow

With top-down compilation, one designer or project lead compiles the entire design in the software. Different designers or IP providers can design and verify different parts of the design, and the project lead can add design entities to the project as they are completed. However, the project lead compiles and optimizes the top-level project as a whole.

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

Bottom-Up & Team-Based Incremental Compilation Flow

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

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

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

The system architect can plan design partitions at the top level and use Quartus II incremental compilation to communicate information to lower-level designers through automatically-generated scripts. The Quartus II option **Generate bottom-up design partition scripts**

automates the process of transferring top-level project information to lower-level modules. The software provides a project manager interface for managing project information in the top-level design.

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

Mixed Incremental Compilation Flow

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

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

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

Early Timing Estimation

It is much less costly to find design issues early in the design cycle than to find problems in the final timing closure stages. Once 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.

Regardless of your compilation flow, when the design source code is complete you can use the **Start Early Timing Estimate** command to perform a quick compilation and timing analysis of your design. The software places any LogicLock regions used to create a floorplan, and finds a quick initial placement for all the design logic, providing a useful estimate of the final design performance.

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

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

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

A system architect can use this methodology along with design partition scripts (as described in "Bottom-Up & Team-Based Incremental Compilation Flow" on page 1–12) to pass additional constraints to lower-level designers, and provide more information about the other partitions in the design. This information can be especially useful to optimize cross-partition paths. Running early timing estimations helps designers find and resolve design problems during the early design stages.

Conclusion

Today's FPGAs support large, complex designs with fast timing performance. By planning several aspects of these FPGAs early in the design process, designers can reduce unnecessary time spent handling these issues in later stages of design. You can use various features of the Quartus II software to quickly plan your design and achieve the best possible results. Choosing the correct device and programming method, planning I/O pin locations, estimating power consumption, performing good design partitioning, and obtaining early timing estimates all improve productivity which reduces the design cost and improves the final product's time to market.

Document Revision History

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

Table 1–1. Documentation Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Initial release.	

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

QII51015-6.1.0

Introduction

For today's high-density, high-performance FPGA designs, the ability to iterate rapidly during the design and debugging stages is critical. The Quartus® II software delivers advanced technology to create designs for high-density FPGAs. Altera® introduced the FPGA industry's first true incremental design and compilation flow, which provides 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 approximately 60%, allowing you to perform more design iterations per day and achieve timing closure more efficiently.
- Provides ease of use through the graphical user interface (GUI).
- Includes Tcl scripting, command-line, and makefile support.
- Facilitates modular hierarchical and team-based design flows using top-down or bottom-up methodologies.
- Supports full incremental compilation for the Stratix® and Cyclone® series of devices, and incremental synthesis for the MAX® II device family. Supports some incremental compilation flows for HardCopy® II devices (for details, refer to "HardCopy Compilation Flows" on page 2–69).

Quartus II incremental compilation is an optional compilation flow. This chapter provides an overview of the Quartus II design flow with and without incremental compilation. However, for an overview of the Quartus II design flow and features, refer to the *Introduction to Quartus II* manual.

This chapter includes the following sections:

To take advantage of incremental compilation, organize your design into logical partitions and physical regions for synthesis and fitting (or placement and routing). Incremental compilation preserves the compilation results and performance of unchanged partitions in your design, dramatically reducing design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. Additionally, you can target optimization techniques, such as physical synthesis, to specific design partitions while leaving other partitions untouched.

In conventional FPGA design, a hierarchical design is flattened into a single netlist before logic synthesis and fitting, and the entire design is recompiled every time the design changes. The Quartus II incremental compilation feature provides the ability to partition a design along any of its hierarchical boundaries. The Quartus II software separately synthesizes and fits each individual hierarchical design partition, then merges the partitions into a complete netlist for subsequent stages of the compilation flow. When recompiling the design, you can choose to use source code, post-synthesis results, or post-fitting results for each partition. If you want to preserve the Fitter results, you can choose to keep just the Fitter netlist, keep the placement results, or keep both the placement and routing results.

Incremental compilation supports two design methodologies: top-down, in which one designer manages the project for the entire design, and bottom-up, in which each design block can be developed independently. Bottom-up methodologies include team-based design flows in which design partitions are created by team members in another location or by third-party intellectual property (IP) providers. For bottom-up flows, you can generate scripts from the top-level design that pass constraints to lower-level design blocks compiled in separate Quartus II projects.

This chapter contains the following information:

- Provides an overview of the Quartus II design flow with and without incremental compilation
- Describes how to use the Quartus II incremental compilation feature

- Provides you the level of understanding required to make good design decisions to achieve timing closure while speeding up design iterations
- Presents several applications of incremental compilation in the form of user scenarios, along with the rationale behind them and the steps required to carry out the tasks

Quartus II Design Flow

Quartus II incremental compilation enhances the standard Quartus II design flow by allowing you to reuse satisfactory results from previous compilations and save compilation time. This section outlines the flat compilation flow with no design partitions and the incremental flow, highlights the differences, and explains some of the reasons you might want to create design partitions and use the incremental flow.

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

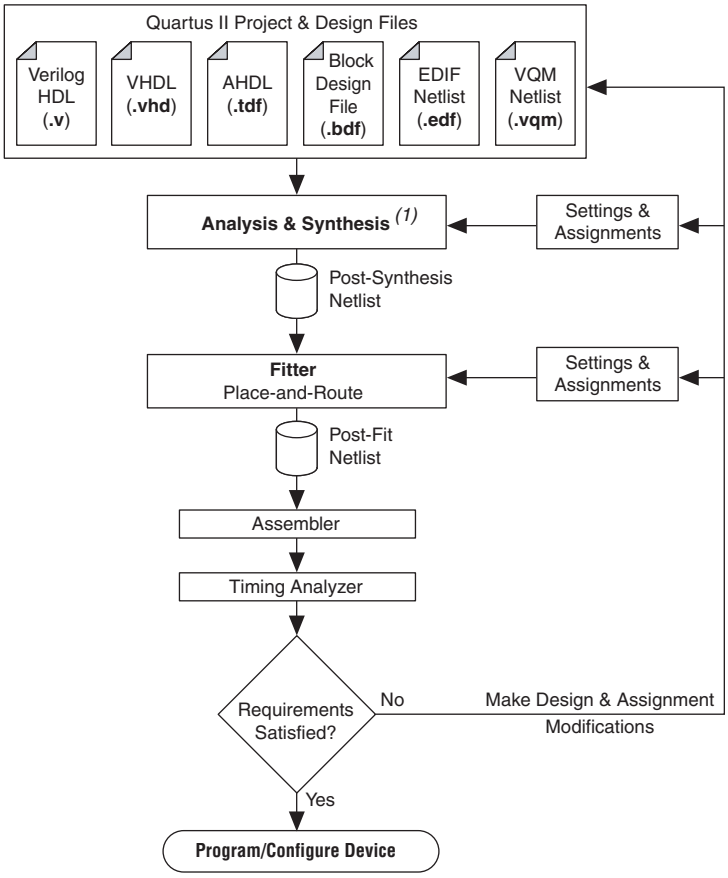
Flat Compilation Flow with No Design Partitions

The standard Quartus II compilation flow consists of the following essential modules:

- **Analysis & Synthesis**—performs logic synthesis to minimize the design logic and performs technology mapping to implement the design logic using device resources such as logic elements. This stage also generates the project database that integrates the design files (including netlists from third-party synthesis tools). When you are using EDIF or VQM netlists created by third-party synthesis tools, the Analysis & Synthesis stage performs logic synthesis and technology mapping only for black boxes and Altera megafunctions.
- **Fitter**—places and routes the logic of a design into a device.
- **Assembler**—converts the Fitter's device, logic, and pin assignments into programming files for the device.
- **Timing Analyzer**—analyzes and validates the timing performance of all the logic in a design.

Figure 2–1 shows a block diagram of the Quartus II design flow with no design partitions.

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



Note to Figure 2–1:

(1) When you are using EDIF or VQM netlists created by third-party EDA synthesis tools, the Analysis & Synthesis stage of the compilation is performed to create the design database, but logic synthesis and technology mapping are performed only for black boxes and Altera megafunctions.

In any Quartus II compilation flow, you can use smart compilation to allow the compiler to determine which compiler modules are required based on the changes made to the design since the last smart compilation, and then skip any modules that are not required. For example, when smart compilation is selected, the compiler skips the Analysis & Synthesis module if the design source files were unchanged. Smart compilation

skips only entire compiler stages. It cannot make incremental changes within a given stage of the compilation flow. To turn on smart compilation, on the Assignments menu, click **Settings**. In the Category list, select **Compilation Process Settings** and click **Use Smart Compilation**.

In the default flat compilation flow, all of the source code is processed with the Analysis & Synthesis module, and all the logic is placed by the Fitter module whenever the design is recompiled after a change in any part of the design. One reason for this behavior is to obtain optimal quality of results. By processing the entire design, the compiler can perform global optimizations to improve area and performance.

Incremental Compilation Flow with Design Partitions

There are many situations in which an incremental compilation flow is more desirable than the simple flat compilation flow. The incremental flow involves splitting your design into blocks called design partitions. Refer to "Design Partitions" on page 2–14 for more details. When the design partitions are well chosen and placed in the device floorplan, you can speed up your design compilation time while maintaining or even improving the quality of results. "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41 provides tips for choosing design partitions.

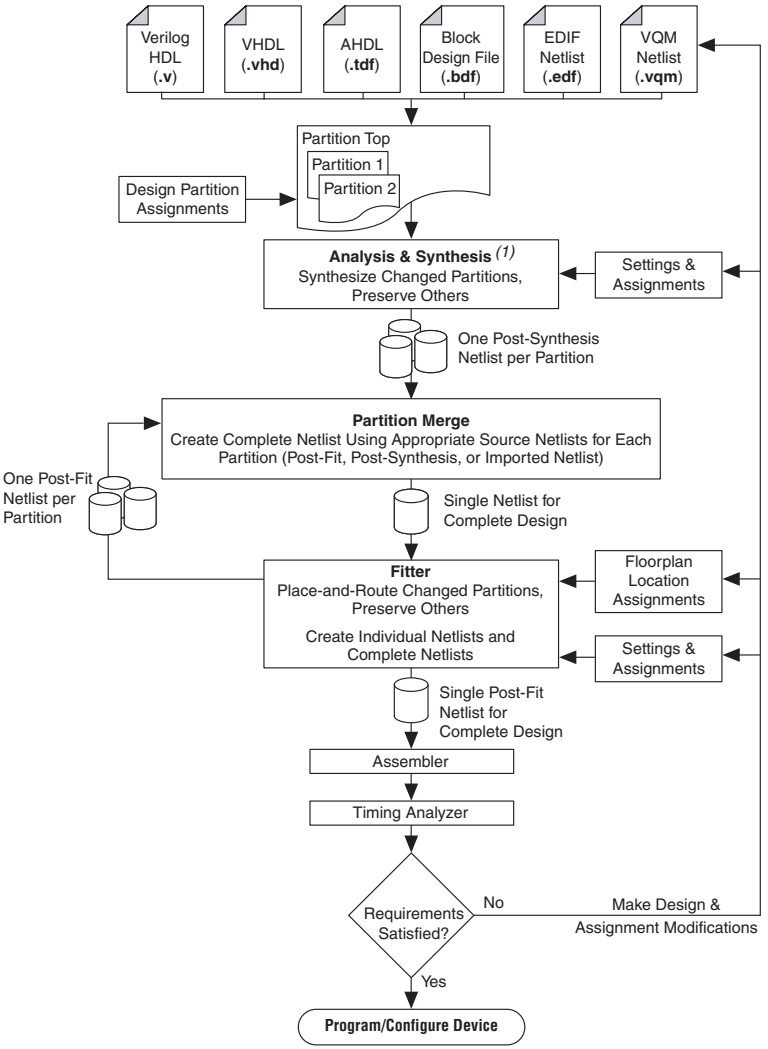
You may want to use incremental compilation later in the design cycle when you are not interested in improving the majority of the design any further, and want to make changes to or optimize one specific block. In this case, you may want to preserve the performance of modules that are unmodified and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

You might also have part of your design that is not yet complete, for which you can create an empty partition while compiling the completed partitions, and 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 using a team-based methodology, and you want to combine them in a bottom-up compilation flow.

For more detailed user scenarios, refer to "User Scenarios—Incremental Compilation Application Examples" on page 2–56.

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

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



Note to Figure 2–2:

(1) When you are using EDIF or VQM netlists created by third-party EDA synthesis tools, the Analysis & Synthesis stage of the compilation is performed to create the design database, but logic synthesis and technology mapping are performed only for black boxes and Altera megafunctions.

In this flow, you partition the design, then perform logic synthesis and technology mapping for each partition individually with Analysis & Synthesis.

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

The partition merge step creates a single, complete netlist that consists of post-synthesis netlists, post-fit netlists, and netlists imported from lower-level projects, depending on the netlist type you specify for each partition.

The Fitter then processes the merged netlist, preserving the placement or placement and routing of unchanged partitions, refitting only those partitions that have changed. The Fitter generates the complete netlist for use in further stages of the compilation flow, including timing analysis and programming file generation. It also generates individual netlists for each partition so that the partition merge step can use the post-fit netlist to preserve the placement and routing of a partition if you specify to do so in future compilations.

If the design does not meet its requirements (functionality, timing, or area), you can make changes to the design and recompile. The Quartus II software does not resynthesize or refit unchanged partitions that have a netlist type assignment that specifies the use of a post-synthesis or post-fit netlist, respectively.

For more information about using the incremental compilation feature, refer to "Preparing a Design for Incremental Compilation" on page 2–10 and "Compiling a Design Using Incremental Compilation" on page 2–11.

Top-Down vs. Bottom-Up Design Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up 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 are changing.

Bottom-up design flows allow individual designers to complete the optimization of their design in separate projects and then integrate each lower-level project into one top-level project. Incremental compilation provides export and import features to enable this design methodology. Designers of lower-level blocks can export the optimized netlist for their design, along with a set of assignments such as LogicLock™ regions. The project lead then imports each design block as a design partition in a top-level project. In this case, the project lead must provide guidance to designers of lower-level blocks to ensure that each partition uses the appropriate device resources.

It is important to realize that with the full incremental compilation flow, users who traditionally relied on a bottom-up approach for the sole reason of performance preservation can now employ a top-down approach to achieve the same goal. This ability is important for two reasons. First, a top-down flow is generally simpler to perform than its bottom-up counterpart. For example, the need to export and import lower-level designs is eliminated. Second, a top-down approach provides the design software with information about the entire design so it can perform global optimizations. In a bottom-up design methodology, you must perform resource balancing and time-budgeting because the software does not have any information about the other partitions in the top-level design when it compiles individual lower-level partitions. For more information about the export and import operations, and how to use design partition scripts to help with design planning, refer to "Exporting & Importing Partitions for Bottom-Up Design Flows" on page 2–26.

Using Incremental Synthesis Only Instead of Full Incremental Compilation

You can turn on incremental compilation for only the synthesis stage of compilation to perform incremental synthesis, with no incremental place-and-route. In this mode, the Fitter uses a flattened netlist without partition boundaries and therefore performs cross-boundary optimizations that can help timing performance. The difference between this flow and the one shown in Figure 2–2 is that the partition merge stage does not accept post-fit netlists produced by the Fitter, and the Fitter does not compile partitions separately.

Full incremental compilation is turned on by default, but you can specify incremental synthesis only. Although the potential benefit offered by full incremental compilation can be higher than that offered by incremental synthesis alone, many additional design considerations and a deeper understanding of the compilation process are required to use the full incremental compilation flow successfully.

Table 2–1 lists the different characteristics between the two compilation options.

Table 2–1. Comparison of Using Incremental Synthesis Only and Full Incremental Compilation

Characteristic	Incremental Synthesis Only	Full Incremental Compilation
Compilation Time Savings	Can save up to 40% of total compilation time; savings limited to Quartus II integrated synthesis.	Typically saves 50-70% of compilation time when post-fit netlists are preserved; savings in both Quartus II integrated synthesis and the Fitter.
Performance Preservation	None since placement cannot be preserved.	Excellent when critical paths are contained within a partition, because you can preserve post-fitting information for unchanged partitions.
Node Name Preservation	Preserves post-synthesis node names for unchanged partitions.	Preserves post-fitting node names for unchanged partitions.
Area Changes	Area might increase because cross-boundary optimizations are no longer possible.	Area might increase by up to 5% over incremental synthesis because placement and register packing are restricted.
f_{MAX} Changes	f_{MAX} might be reduced because cross-boundary optimizations are no longer possible. If the design is partitioned appropriately, there is no negative impact on f_{MAX}.	f_{MAX} might be reduced because cross-boundary optimizations are no longer possible. If the design is partitioned and the floorplan location assignments are created appropriately, no negative impact on f_{MAX}. You may get a slight performance increase by employing a good partition and floorplan scheme.
Ease of Use	Simply create partitions.	Specify which partitions you want to preserve, and create floorplan location assignments in most cases.
Floorplan Creation	Not required.	Required in most cases for best quality of results.
When Design is Resynthesized	Any time you make a change to the source code or synthesis assignments.	Any time you make changes to the source code, unless you specify a Post-Fit (Strict) netlist, or it is an imported partition. When you specify to use the source file.
When Design is Refit	Every time you perform incremental synthesis.	Any time you make changes to the source code, unless you specify a Post-Fit (Strict) netlist, or it is an imported partition. When you specify to use the source, the post-synthesis netlist, or the post-fit netlist with a Fitter preservation level of Netlist Only.

For usage details specific to the **Incremental synthesis only** option, refer to the *Incremental Synthesis* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Preparing a Design for Incremental Compilation

To set up your design for incremental compilation, use the following general steps. Detailed descriptions for some of these steps are included in later sections of this chapter. The flow chart in Figure 2–3 on page 2–12 illustrates these steps in the complete incremental design flow.

1. Elaborate the design. On the Processing menu, point to Start and click **Start Analysis & Elaboration**, or run any compilation flow that includes this step. This allows the Quartus II software to identify your design's hierarchy.

2. Ensure that incremental compilation is turned on (it is on by default). If required, on the Assignments menu, click **Settings**. In the **Settings** dialog box, select **Incremental Compilation** in the Compilation Process Settings folder and turn on **Full incremental compilation**.

3. Create partitions in your design by applying the **Set as Design Partition** or `PARTITION_HIERARCHY` assignment to the appropriate instances.

   Refer to "Design Partitions" on page 2–14 for an explanation of design partitions and what part of your design can be specified as a design partition. Refer to "Creating Design Partitions" on page 2–16 for more details about assigning design partitions. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41.

   When you specify your first partition, a dialog box is shown that asks whether you want to enable incremental compilation if it is not already turned on. Selecting **Full incremental compilation** in this dialog box also turns on incremental compilation as in Step 2. You can also turn on incremental compilation in the **Design Partitions Window** on the Assignments menu.

   Selecting **Off** on the Incremental Compilation page of the **Settings** dialog box turns off all forms of incremental synthesis and incremental compilation, but does not remove any partition assignments. Partition assignments have no effect on the design if incremental compilation is turned off.

4. Use LogicLock regions to make location assignments for each partition in the design to create a design floorplan. Each partition should be assigned to a physical region on the device. Refer to the section "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–24 for details on making these assignments. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41.

5. On the Processing menu, click **Start Compilation** to compile the design. The first compilation after making the partition and LogicLock assignments is a complete compilation that prepares the design for subsequent incremental compilations.

To use incremental synthesis only, follow Step 1 through Step 3, but in Step 2 turn on **Incremental synthesis only** instead of **Full incremental compilation**.

For details about using only the incremental synthesis option, refer to the *Incremental Synthesis* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Compiling a Design Using Incremental Compilation

After compiling the design once and then making changes, you can take advantage of incremental compilation to recompile the changed parts of the design while preserving the results for the unchanged partitions, thus saving time on subsequent compilations. To do this, perform the following general steps:

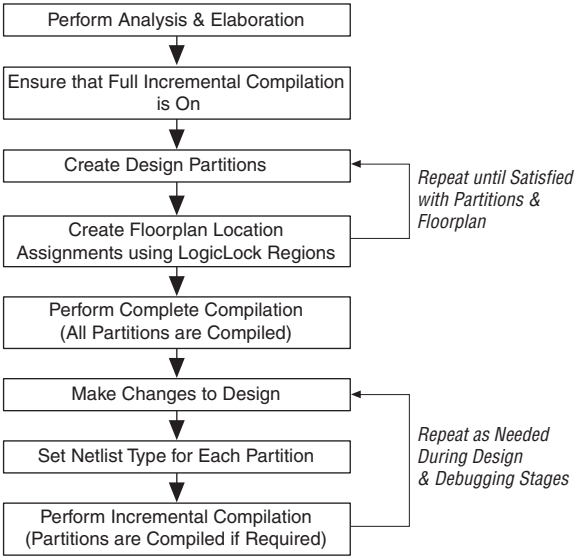
1. To preserve previous compilation results for a partition, set the **Netlist Type** assignment for that partition to **Post-Fit**. To save just the synthesis results, set the **Netlist Type** assignment for that partition to **Post-Synthesis.** If you have imported a partition from another Quartus II project or from an exported partition in the current project, choose **Imported**. For details on setting this partition property and specifying the Fitter preservation level for post-fit netlists, refer to "Setting the Netlist Type for Design Partitions" on page 2–19.

2. Compile the design. When you start a compilation for a partitioned design with incremental compilation turned on, the Quartus II software automatically uses the incremental compilation flow, preserving the results as specified in Step 1.

If you preserve the compilation results using the Post-Fit netlist, you do not have to back-annotate any logic location assignments. You should not use the incremental compilation and the assignment back-annotation features in the same Quartus II project.

The flow chart in Figure 2–3 illustrates these steps in the complete incremental design flow.

Figure 2–3. Summary of Design Flow Using Incremental Compilation



For bottom-up and team-based design flows, the designers of lower-level design blocks must export their designs so the top-level designer can import them into the top-level design. Refer to "Exporting & Importing Partitions for Bottom-Up Design Flows" on page 2–26 for details.

What Represents a Source Change for Incremental Compilation?

The Quartus II software uses an internal checksum to determine whether the contents of a source file have changed. Source files are the design files used to create the design, and consist of VHDL files, Verilog HDL files, AHDL files, Block Design Files (**.bdf**), EDIF netlists, VQM netlists, and memory initialization files. Changes in other files such as vector waveform files for simulation do not trigger recompilation.

Synthesis and Fitter assignments, including optimization settings, timing assignments, or Fitter location assignments such as pin assignments or LogicLock 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 re-run routing using existing placement results except for any new routing settings including delay chain settings

For information about the **Netlist Type** and **Fitter Preservation Level** assignments, refer to "Setting the Netlist Type for Design Partitions" on page 2–19.

The project database folder (**\db**) includes all the netlist information for previous compilations. To avoid unnecessary recompilations, the database files must not be altered or deleted.

If you want to archive or reproduce the project in another location, you can use a Quartus II Archive (**.qar**) file. On the Project menu, click **Archive Project** and turn on **Include database from compilation and simulation** so that compilation results are preserved. To manually create a project archive that preserves compilation results without keeping the entire compilation database, you should keep all source and settings files and create and save a Quartus II Exported Partition (**.qxp**) file for each partition in the design. Refer to "Exporting a Lower-Level Block within a Project" on page 2–30 for more details.

Determining Which Partitions Will Be Recompiled

When design files in a partition have dependencies on other files, changing one file may trigger an automatic recompilation of another file. The **Partition Dependent Files** table in the Analysis & Synthesis report lists the design files that contribute to each design partition. You can use this table to determine which partitions will be recompiled when a specific file is changed.

For example, if a design has files **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**, then the entities in file **a.v** depend on files **b.v** and **c.v,** so files **b.v** and **c.v** are listed in the report table as dependent files for the partition containing entity **a**.

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

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

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

Forcing the use of the post-fitting netlist when the contents of a source file has changed is recommended only for advanced users who thoroughly understand when a partition must be recompiled. To force the Fitter to use a previously generated post-fit netlist when there are changes to the source files, you can use the **Post-Fit (Strict)** Netlist Type assignment. For information about the **Post-Fit (Strict)** Netlist Type, refer to "Setting the Netlist Type for Design Partitions" on page 2–19.

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

Design Partitions

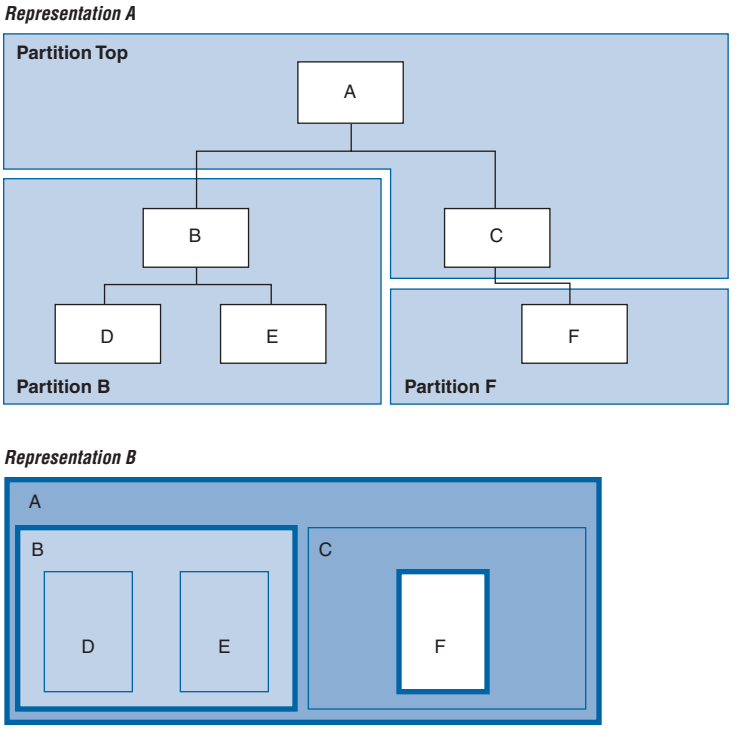
A common design practice is to create modular or hierarchical designs in which you develop each design entity separately and then instantiate them in a higher-level entity, forming a complete design. The software does not consider each design entity automatically to be a design partition for incremental compilation; rather, you must designate one or more design hierarchies below the top-level project to be a design partition. Creating partitions prevents the compiler from performing optimizations across partition boundaries, as discussed in "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41 and illustrated in Figure 2–10. 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 partitions cannot be a portion of the logic within a hierarchical entity. When you declare a partition, every hierarchical entity within that partition becomes part of the same partition. You can create new partitions for hierarchical entities within an existing partition, in which case the entities within the new partition are no longer included with the higher-level partition, as described in the following example.

In Figure 2–4, hierarchical entities **B** and **F** form partitions in the complete design, which is made up of entities **A**, **B**, **C**, **D**, **E**, and **F**. The shaded boxes in Representation A indicate design partitions in a "tree" representation of the hierarchy. In Representation B, the lower-level entities are represented inside the higher-level entities, 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 entities below it, or it may contain its own logic. In this example, the partition for top-level entity **A** also includes the logic in one of its lower-level entities, **C**. Because entity **F** is contained in its own partition, it is not treated as part of the top-level partition. Another separate partition, **B**, contains the logic in entities **B**, **D**, and **E**.

Figure 2–4. Partitions in a Hierarchical Design



Design Partitions Compared to Physical Regions

Design partitions for incremental compilation are logical partitions, different from physical regions in the device floorplan. Physical regions specify locations in the device floorplan using LogicLock assignments in the Quartus II software. Physical regions have a size and location on the device floorplan, and you can assign multiple design instances and nodes to a physical region to place them close to each other. A logical design

partition does not refer to a physical section of the device and does not directly control the placement of instances. A logical design partition sets up a virtual boundary between design hierarchies so each is compiled separately, preventing logical optimizations from occurring between them.

Altera recommends that you assign each design partition to a physical region using LogicLock assignments to improve the quality of results when performing a full incremental compilation. Create floorplan location assignments for design partitions using LogicLock regions as discussed in "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–24. Physical location assignments are not required for logical design partitions if you are using the **Incremental Synthesis Only** option.

Creating Design Partitions

To use incremental compilation, you must first split your design into partitions, as described in "Design Partitions" on page 2–14 and "Preparing a Design for Incremental Compilation" on page 2–10. You can make partition assignments to HDL or schematic design instances, or to VQM or EDIF netlist instances (from third-party synthesis tools). To take advantage of incremental compilation when source files change, the top-level design entity of each partition should have a unique design file. If you define two different entities of 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.

When you are using a third-party synthesis tool, create a separate netlist file for each partition to allow each partition to be treated incrementally. To create separate netlists for each partition, you may have to create a top-level HDL wrapper file that instantiates the lower-level netlist files and then create separate projects in your synthesis tool for each of the lower-level partitions. In this case, the lower-level blocks should be treated as a black box in the top-level design. Some synthesis tools allow you to create separate netlist files for different design blocks within a single project.

For information about using incremental compilation with third-party synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

For suggestions on determining which parts of your design should be set as design partitions, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41.

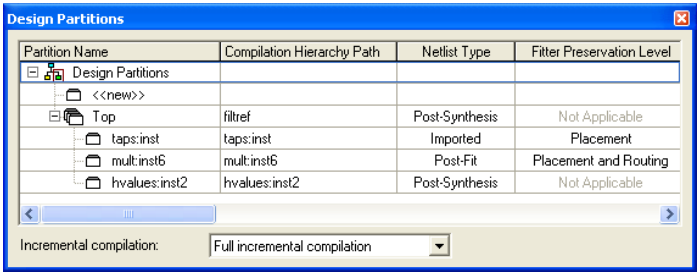
You can create design partitions in the Quartus II GUI with the Design Partitions Window or the Project Navigator.

On the Assignments menu, click **Design Partitions Window** (Figure 2–5) to create your partitions in one of the following ways:

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

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

Figure 2–5. Design Partitions Window

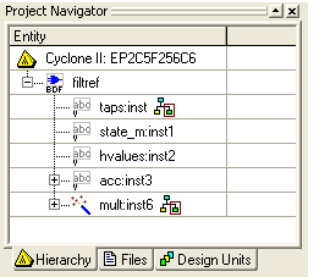


Alternatively, you can use the list of instances under the **Hierarchy** tab in the **Project Navigator** to create and delete design partitions. Right-click on an instance in the **Project Navigator** and click **Set as Design Partition**.

A design partition icon appears next to each instance that is set as a partition (Figure 2–6).

To remove an existing partition assignment, right-click the instance in the **Project Navigator** and click **Set as Design Partition** again. (This process turns off the option.)

Figure 2–6. Project Navigator Showing Design Partitions



Partition Name

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

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

Setting the Netlist Type for Design Partitions

The **Netlist Type** property controls the incremental compilation process, as described in "Compiling a Design Using Incremental Compilation" on page 2–11. The **Netlist Type** is a property of each design partition that allows you to specify the type of netlist or source file that the compiler should use as the input for each partition. This property determines which netlist is used by the Partition Merge stage in the next compilation.

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

Table 2–2 describes the different settings for the **Netlist Type** property, explains the behavior of the Quartus II software for each setting, and gives guidance on when to use a certain setting.

Table 2–2. Netlist Type Settings (Part 1 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Source File	Always compiles the partition using the associated design source file(s). You can use this netlist type to recompile a partition from the source code using new synthesis or Fitter settings. If a partition has an associated imported netlist, compiling it with netlist type set to Source File removes the imported netlist.
Post-Synthesis	Preserves post-synthesis results for the partition and uses the post-synthesis netlist as long as the following conditions are true: • A post-synthesis netlist is available from a previous synthesis • No change has been made to the associated source files since the previous synthesis Compiles the partition from the source files if there are source changes or if a post-synthesis netlist is not available. Changes to the assignments do not cause recompilation. You can use this netlist type to preserve the synthesis results unless source files change, but refit the partition using any new Fitter settings. If a partition has an associated imported netlist, this setting is not available.

Table 2–2. Netlist Type Settings (Part 2 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Post-Fit	Preserves post-fit results for the partition and uses the post-fit netlist as long as the following conditions are true: ● A post-fit netlist is available from a previous fitting ● No change has been made to the associated source files since the previous fitting Compiles the partition from the source files if there are source changes or if a post-fit netlist is not available. Changes to assignments do not cause recompilation. The **Fitter Preservation Level** specifies what type of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level" on page 2–21. You can use this netlist type to preserve the Fitter results unless source files change. You can also use this netlist type to apply global optimizations, such as Physical Synthesis optimizations, to certain partitions while preserving the fitting results for other partitions. If a partition has an associated imported netlist, this setting is not available.
Post-Fit (Strict)	Always preserves post-fit results for the partition. Uses the post-fit netlist even if changes have been made to the associated source files since the previous fitting. The **Fitter Preservation Level** specifies what type of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level" on page 2–21. If a partition has an associated imported netlist, this setting is not available.
Imported	Compiles the design partition using a netlist imported from a Quartus II Exported Partition File **(.qxp)**. The software does not modify or overwrite the original imported netlist during compilation. To preserve changes made to the imported netlist (such as movement of an imported LogicLock region), use the **Post-Fit (Import-based)** setting following a successful compilation with the imported netlist. For additional details, refer to "Exporting & Importing Partitions for Bottom-Up Design Flows" on page 2–26. The **Fitter Preservation Level** specifies what type of information is preserved from the imported netlist. For details, refer to "Fitter Preservation Level" on page 2–21. If you have not imported a netlist for this partition using the Import Design Partition command, this setting is not available.

Table 2–2. Netlist Type Settings (Part 3 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Post-Fit (Import-based)	Preserves post-fit results for the partition and uses the post-fit netlist as long as the following conditions are true: ● A post-fit netlist is available from a previous fitting ● No change has been made to the associated imported netlist since the previous fitting Compiles the partition from the imported netlist if the imported netlist changes (which means it has been reimported) or if a post-fit netlist is not available. Changes to assignments do not cause recompilation. The **Fitter Preservation Level** specifies what type of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level". You can use this netlist type to preserve changes to the placement and routing of an imported netlist. If a partition does not have an associated imported netlist, this setting is not available.
Empty	Uses an empty placeholder netlist for the partition and uses virtual pins at the partition boundaries. You can use this netlist type to skip the compilation of a lower-level partition. For more details on the Empty setting, refer to "Empty Partitions" on page 2–23.

Fitter Preservation Level

The Fitter Preservation Level property specifies which information the compiler will use from a post-fit or imported netlist. The property is only available if the **Netlist Type** is set to **Post-Fit**, **Post-Fit (Strict)**, **Imported**, or **Post-Fit (Import-based)**.

On the Assignments menu, click **Design Partitions Window**. You can view and modify the **Fitter Preservation Level** by double-clicking an entry. You can also right-click and click **Properties**, then edit the **Fitter Preservation Level** on the **Compilation** tab.

Table 2–3 describes the Fitter Preservation Level settings.

Table 2–3. Fitter Preservation Level Settings

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Placement	Preserves the netlist atoms and their placement in the design partition. Re-routes the design partition. This setting saves significant compilation time because the Fitter does not need to re-fit the nodes in the partition. Note that the Fitter may need to modify the placement for timing or legality reasons. This setting might not be available if the netlist type is set to **Imported** and the imported netlist does not contain placement data.
Placement and Routing	Preserves the netlist atoms and their placement and routing in the design partition. Required to preserve Engineering Change Order (ECO) changes made to the post-fitting netlist and SignalProbe pins added to the design. This setting minimizes compilation time. Note that the Fitter may need to modify the placement and routing for timing or legality reasons. This setting may not be available if the netlist type is set to **Imported** and the imported netlist does not contain routing data.
Netlist Only	Preserves the netlist atoms of the design partition, but replaces and re-routes the design partition. Unlike a Post-Synthesis netlist, a Post-Fit netlist with the atoms preserved contains any Fitter optimizations, for example, registers duplicated by Physical Synthesis 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 (that is, physical synthesis) that would otherwise be impossible when the placement is locked down ● Resolve resource conflicts between two imported partitions in a bottom-up design flow

Empty Partitions

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

You can use the **Empty** setting to skip the compilation of a lower-level partition that is incomplete or missing from the top-level design. You can also use it if you want to compile only some partitions in the design, such as during optimization or if the compilation time is large for one partition and you want to exclude it.

When a partition **Netlist Type** is defined as **Empty**, virtual pins are created at the boundary of the partition. This means that the software temporarily maps I/O pins in the lower-level design entity to internal cells and not to pins during compilation.

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

You can use a design flow in which some partitions are set to **Empty** in a variation of a bottom-up design flow, where you develop pieces of the design separately and then combine them at the top level at a later time. When you implement part of the design without information about the rest of the project, it is impossible for the Compiler to perform global placement optimizations. One way to reduce this effect is to ensure the input and output ports of the partitions are registered whenever possible, as recommended in "Creating Good Design Partitions" on page 2–42.

When you set a design partition to **Empty**, a design file is required in Analysis & Synthesis to specify, at minimum, the port interface information so that it can connect the partition correctly to other logic and partitions in the design. If the design file is missing, you must create a wrapper file (called a black box or hollow-body file) that defines the design block and specifies the input, output, and bidirectional ports. For example, in Verilog HDL you should include a module declaration, and in VHDL you should include an entity and architecture declaration.

Creating a Design Floorplan With LogicLock Location Assignments

After you have partitioned the design, create floorplan location assignments for the design as discussed in this section to improve the quality of results when using the full incremental compilation flow.

The simplest way to create a floorplan for a partitioned design is to create one LogicLock region per partition (including the top-level partition). Initially, you can leave each region with the default settings of Auto size and Floating location to allow the Quartus II software to determine the optimal size and location for the regions. Then, after compilation, back-annotate the Fitter-determined size and location properties to use 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.

For more information on why creating a design floorplan is important, refer to "The Importance of Floorplan Location Assignments in Incremental Compilation" on page 2–50. For guidelines on creating the floorplan, refer to "Creating Good Floorplan Location Assignments" on page 2–52.

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

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

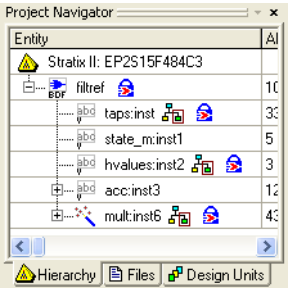
2. Create a LogicLock region for each partition (including the top-level entity, which is automatically considered a partition) using one of the following methods:

   - In the Design Partitions window, right-click on a partition and click **Create New LogicLock Region**. You can highlight multiple (or all) partitions by holding down the **Ctrl** key and clicking on each partition. Then you can choose the option to create a separate LogicLock region for each highlighted partition.
   - Under **Compilation Hierarchy** in the **Project Navigator**, right-click each instance that is denoted as a partition and click **Create New LogicLock Region**.

   A LogicLock icon appears in the Project Navigator next to each instance that is set as a LogicLock region (Figure 2–7).

Figure 2–7. Project Navigator Showing LogicLock Regions



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

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

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

4. On the Assignments menu, click **LogicLock Regions Window**, and click on each LogicLock region while holding the Ctrl key to select all regions (including the top-level region).

5. Right-click on the last selected LogicLock region, and click **Properties**.

6. On the **Location** tab, click **Back-annotate Origin and Lock** to back-annotate the Fitter-determined size and location properties, then click **OK**.

   It is important that you use the Fitter-chosen locations only as a starting point to make the regions of a fixed size and location. Regions with fixed size and location yield better f_{MAX} than auto-sized regions on average.

   Do not back-annotate the contents of the region, just save the location and size using the **Back-annotate Origin and Lock** command. Placement is preserved through the use of the post-fit netlist instead of back-annotated content assignments.

7. If required, modify the size and location via the **LogicLock Regions Window** or the **Timing Closure Floorplan**.

8. On the Processing menu, point to Start and click **Start Early Timing Estimate** to estimate the timing performance of your design with these LogicLock regions.

9. Repeat steps 7 and 8 until you are satisfied with the quality of results for your design floorplan. On the Processing menu, click **Start Compilation** to run a full compilation.

Taking Advantage of the Early Timing Estimator

The methodology for creating a good floorplan takes advantage of the Early Timing Estimator to enable quick compilations of the design while creating assignments. The Early Timing Estimator feature provides a timing estimate for a design as much as 45 times faster than running a full compilation, yet estimates are, on average, within 11% of final design timing. You can use the Timing Closure Floorplan editor to view the "placement estimate" created by this feature, identify critical paths, 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.

Exporting & Importing Partitions for Bottom-Up Design Flows

The bottom-up flow refers to the design methodology in which a project is first divided into smaller subdesigns that are implemented as separate projects, potentially by different designers. The compilation results of these lower-level projects are then exported and given to the designer (or the project lead) who is responsible for importing them into the top-level project to obtain a fully functional design.

There are at least two benefits associated with a bottom-up design flow:

- It facilitates team-based development.
- It permits the reuse of compilation results from another project, with the ultimate goals of performance preservation and compilation time reduction.

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

- It may be difficult to achieve timing closure for the full design, because you compile the lower-level submodules independently without any information about each other. This problem may be avoided by careful timing budgeting and special design rules such as always registering the ports at the module boundaries.

- For the same reason, resource budgeting and allocation may be needed to avoid resource conflicts and overuse. Floorplan creation is typically very important in a bottom-up flow.

In a bottom-up design flow, the top-level project lead can do much of the design planning, and then pass constraints on to the designers of lower-level blocks. The bottom-up design partition scripts generated by the Quartus II software can make it easier to plan a bottom-up design, and limit the difficulties that can arise when integrating separate designs. For examples of team-based scenarios, refer to "Bottom-Up Incremental Design Flow Scenarios" on page 2–60.

Preparing the Top-Level Design for a Bottom-Up Incremental Compilation Methodology

To set up your design for bottom-up incremental compilation, use the following general steps:

1. Create a top-level project that will be compiled by the project lead and will eventually incorporate the entire design. The top-level design file must include the top-level entity that instantiates all the lower-level subdesigns that you plan to compile in separate Quartus II projects and import as separate design partitions.

2. In your top-level project, include a wrapper design file for each subdesign partition that defines at least the port interface of the subdesign. Analysis & Elaboration requires this wrapper file (also known as a "stub" or "black box" file) to connect all the separate design partitions at the top level. The wrapper file must contain the entity definition, including the correct port list. For example, in Verilog HDL you should include a module declaration, and in VHDL you should include an entity and architecture declaration. The wrapper file does not have to contain the logic for the design partition.

3. Create all global assignments, including the device assignment, pin location assignments, and timing assignments, so that the final design meets its requirements. Lower-level project designers can add their own constraints for their partitions as needed, and later export them to top-level, but the basic constraints can be passed down from the top-level to avoid any conflicts and ensure that lower-level projects use the correct assignments.

4. Set up the top-level design with design partitions, turn on incremental compilation, and create a design floorplan using LogicLock assignments. Follow the steps in "Preparing a Design for Incremental Compilation" on page 2–10.

5. Ensure that you allocate device resources appropriately, as described in "Resource Balancing" on page 2–46.

6. Optionally, you can use the Quartus II software to generate bottom-up design partition scripts that help with design planning and project management at the top level of the project. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–35 for details.

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

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

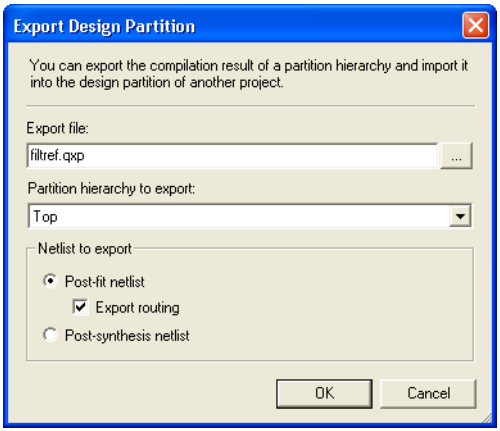
- Ensure that the LogicLock region uses only the resources allocated by the top-level project lead.
- Ensure that you know which clocks should be allocated to global routing resources so that there are no resource conflicts in the top-level design.
  - Set the **Global Signal** assignment to **On** for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

incorporated in the top-level design. You can also apply timing assignments to the I/O ports of the subdesign to perform timing budgeting as described in "Timing Budgeting" on page 2–48.

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

1. In the subdesign project, on the Project menu, click **Export Design Partition**. The **Export Design Partition** dialog box appears (Figure 2–8).

Figure 2–8. Export Design Partition Dialog Box



2. In the **Export file** box, type the name of the Quartus II Exported Partition file (**.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 result of any partition hierarchy in the project, as described in "Exporting a Lower-Level Block within a Project" on page 2–30. Choose the partition hierarchy from the drop-down box.

4. Under **Netlist to export**, select either **Post-fit netlist** or **Post-synthesis netlist**. The default is **Post-fit netlist**. For post-fit netlists, turn on or off the **Export routing** option as required.

5. Click **OK**. The Quartus II software creates the Quartus II Exported Partition (**.qxp**) file in the specified directory.

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

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under **Compilation Process Settings**, select the **Incremental Compilation** page.
3. Turn on **Automatically export design partition after compilation**.
4. If you want to view or change the default export settings, click the **Export Design Partition Settings** button to open the **Export Design Partition Settings** dialog box (Figure 2–8).
5. In the **Export Design Partition Settings** dialog box, change the settings, if required, as in steps 2-4 in the preceding export procedure. Click **OK**.
6. Click **OK** to close the **Settings** dialog box. During the next full compilation, the software will create the Quartus II Exported Partition (**.qxp**) file in the specified directory.

Exporting a Lower-Level Block within a Project

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

You can use this feature to add test logic around a lower-level block that will be exported as a design partition for a top-level design. With this functionality, you can also instantiate other design components in a lower-level project to match the top-level design environment. The project then more accurately captures timing and resource requirements, but you don't have to export all these components to the top-level design.

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

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

The import process involves importing the design netlist from the Quartus II Exported Partition file and adding the netlist to the database for the top-level project. Importing also filters the assignments from the subdesign and creates the appropriate assignments in the top-level project.

To import a subdesign partition into a top-level design:

1. In the top-level project, on the Project menu, click **Import Design Partition**. Alternatively, right-click on the partition that you want to import in the Design Partitions window and click **Import Design Partition**. The **Import Design Partition** dialog box appears.

2. In the **Partition(s)** box, browse to the desired partition. To choose a partition, highlight the partition name in the **Select Partition(s)** dialog box and use the appropriate buttons to select or deselect the desired partitions.

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

3. Under **Import file**, type the name of the Quartus II Exported Partition file or browse for the file that you want to import into the selected partition. Note that this file is required only during importation, and is not used during subsequent compilations unless you re-import the partition.

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

4. To view the contents of the selected Quartus II Exported Partition file, click **Load Properties**. The properties displayed include the Netlist Type, Entity name, Device, and statistics about the partition size and ports.

5. Click **Advanced Import Settings** and make selections, as appropriate, to control how assignments and regions are integrated from a subdesign into a top-level design partition. During importation, some regions may be resized or slightly moved. Click **OK** to apply the settings.

   For more information about the advanced settings, refer to "Importing Assignments & Advanced Import Settings" on page 2–32.

6. In the **Import Design Partition** dialog box, click **OK** to start importation. The specified Quartus II Exported Partition file is imported into the database for the current top-level project.

Importing Assignments & Advanced Import Settings

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

Design Partition Properties After Importing

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

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

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

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

Importing Design Partition Assignments Within the Subdesign

Design partitions defined within the subdesign project are currently not imported to the top-level.

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. If you want to preserve changes made to a partition after compilation, use the Netlist Type **Post-Fit (Import-Based)**.

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

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

Importing Other Instance Assignments

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

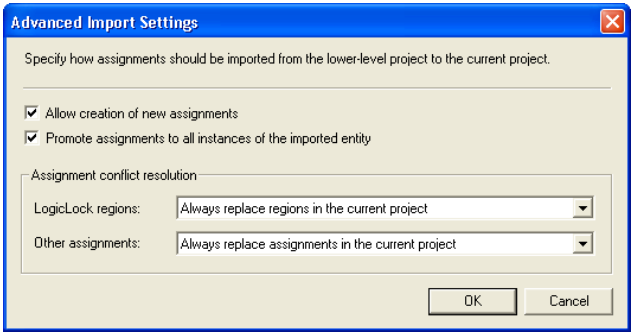
Importing Global Assignments

Global assignments are not imported. The project lead should make global assignments, such as clock settings, in the top-level design.

Advanced Import Settings

The **Advanced Import Settings** dialog box, shown in Figure 2–16, allows you to specify the options that control how assignments and regions are integrated and how to resolve assignment conflicts when importing a subdesign partition into a top-level design. The following subsections describe each of these options.

Figure 2–9. Advanced Import Settings Dialog Box



Allow Creation of New Assignments
Allows the import command to add new assignments from the imported project to the top-level project.

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

Promote Assignments to all Instances of the Imported Entity
Converts and promotes entity-level assignments from the subdesign into instance-level assignments in the top-level design.

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

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

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

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

Generating Bottom-Up Design Partition Scripts for Project Management

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

Generating bottom-up design partition scripts is optional in any bottom-up design methodology.

For example design scenarios using these scripts, refer to "Bottom-Up Incremental Design Flow Scenarios" on page 2–60. In a typical bottom-up design flow, the project lead must perform some or all of the following tasks to ensure successful integration of the subprojects:

- Manually determine which assignments should be propagated from the top level to the bottom levels. This requires detailed knowledge of which Quartus II assignments are needed to set up low-level projects.
- Manually communicate the top-level assignments to the low-level projects. This requires detailed knowledge of Tcl or other scripting languages to efficiently communicate project constraints.
- Manually determine appropriate timing and location assignments that will help overcome the limitations of bottom-up design. This requires examination of the logic in the lower levels to determine appropriate timing constraints.
- Perform final timing closure and resource conflict avoidance at the top level. Because the low-level projects have no information about each other, meeting constraints at the lower levels does not guarantee

they will be met when integrated at the top-level. It then becomes the project lead's responsibility to resolve the issues, even though information about the low-level implementation may not be available.

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

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

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

For information about current limitations in the bottom-up partition scripts, refer to "Bottom-Up Design Partition Script Limitations" on page 2–70.

Project Creation

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

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

Excluded Partitions

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

Assignments from the Top-Level Design

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

The names of the design files that apply to the specific partition are added to each lower-level project. Note that the script uses the file name(s) specified in the top-level project. If the top-level project used a placeholder wrapper file with a different name than the design file in the lower-level project, you should be sure to add the appropriate file to the lower-level project.

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

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

- **Timing assignments**—When this option is turned on, all global timing assignments for the lower-level projects are included in the script, including t_{CO}, t_{SU}, and f_{MAX} constraints. This option may also include timing constraints on internal partition connections.
- **Design partition assignments**—When this option is turned on, script assignments related to design partitions in the lower-level projects are included, as well as assignments associated with LogicLock regions.
- **Pin location assignments**—When this option is turned on, all pin location assignments for lower-level project ports that connect to pins in the top-level design are included in the script, controlling the overuse of I/Os at the top-level during the integration phase and preserving placement.

Virtual Pin Assignments

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

Virtual Pin Timing & Location Assignments

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

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

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

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

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

LogicLock Region Assignments

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

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

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

Global Signal Promotion Assignments

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

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

When **Disable automatic global promotion in lower-level projects** is turned on, the script includes assignments that turn off all automatic global promotion settings in the lower-level projects. These settings

include the Auto Global Memory Control Signals logic option, output enable logic options, and clock and register control promotions. If you select the **Disable automatic global promotion in lower-level projects** option in conjunction with the **Promote top-level global signals in lower-level projects** option, you can ensure that only signals promoted to global resources in the top-level are promoted in the lower-level projects.

Makefile Generation

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

The makefiles use the directory locations generated using the **Create lower-level project if one does not exist** option. If you created your lower-level projects without using this option, you must modify the variables at the top of the makefile to specify the directory location for each lower-level project.

To run the makefiles, use a command such as

```
make -f master_makefile.mak
```

from the script output directory. The master makefile first runs each lower-level makefile, which sources its Tcl script and then generates a Quartus II Exported Partition file to export the project as a design partition. Next, run the top-level makefile that specifies these newly generated Quartus II Exported Partition files as the import files for their respective partitions in the top-level project. The top-level makefile then imports the lower-level results and performs a full compilation, producing a final design.

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

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

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

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

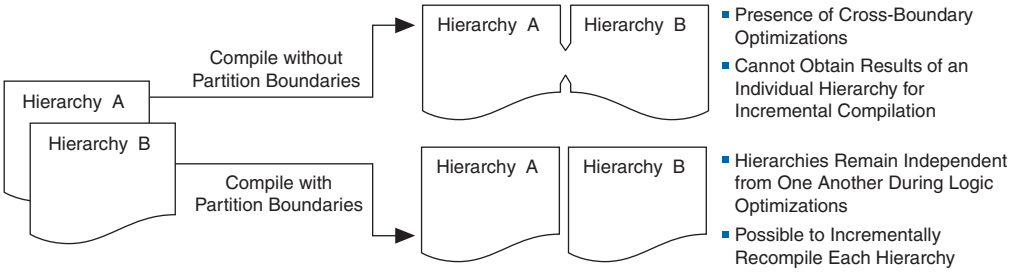
Guidelines for Creating Good Design Partitions and LogicLock Regions

This section provides guidelines for creating design partitions and floorplan location assignments that will help you achieve good quality results, as well as criteria and methodologies to check the quality of your assignments.

When planning your design, keep in mind the size and scope of each partition, and the likelihood that different parts of your design might change as your design develops.

Creating partitions prevents the compiler from performing logic optimizations across partition boundaries (Figure 2–10), allowing the software to synthesize and place each partition separately.

Figure 2–10. Effects of Partition Boundaries During Optimization



Because cross-boundary optimizations cannot occur when using partitions, the quality of results and performance of the design may decrease as the number of partitions increases. Although more partitions allows for greater reduction in compilation time, you should limit the number of partitions to prevent degradation of the quality of results. This effect is more pronounced when using full incremental compilation than when using incremental synthesis only, and can have more effect in a bottom-up methodology than a top-down methodology.

In a top-down compilation where partitions are not locked down with post-fitting results, the Fitter can perform placement optimizations on the design as a whole to optimize the placement of cross-partition paths. (However, the Fitter cannot perform logic optimizations such as physical synthesis across the partition boundary.) In a bottom-up flow, partitions are compiled separately. Typically, the fitting results are exported, so there is no placement optimization across the partitions boundaries.

Creating Good Design Partitions

Altera recommends that you observe the following important hierarchical design considerations when creating partitions:

- Register all inputs and outputs of each partition. This helps avoid any delay penalty on signals that cross partition boundaries. At the very least, either the inputs or the outputs should be registered. The Statistics reports described in the "Partition Statistics Reports" section list the ports registered for each partition.

  While this can be difficult in practice, adherence to this principle results in less timing degradation and area increase when using incremental flows. Registering lessens the need for the cross-partition optimizations that are prevented by partitioning. By registering the ports, you can keep critical paths within a single partition, thus keeping the lengths of inter-partition register-to-register paths to a minimum.

- Minimize the number of paths that cross partition boundaries. If there are critical paths crossing between partitions, rework the partitions to avoid these inter-partition paths. Capturing 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. The Statistics reports described in "Partition Statistics Reports" on page 2–44 list the number of input and output ports for each partition.
- Ensure that the size of each partition is not too small (for example, not less than 1,000 logic elements (LEs) or adaptive logic modules (ALMs)). The Statistics reports described in the "Partition Statistics Reports" section list the logic utilization of each partition.

- Minimize the number of unconnected ports at partition boundaries. When a port is left unconnected, optimizations that remove logic driving that port could improve results. However, these optimizations are not allowed in an incremental design, because they would lead to cross-partition dependence. Altera recommends that you either connect such ports to an appropriate node or remove them from the design. If you know the port should not be used, consider defining a wrapper module with a port interface that reflects this fact. The Statistics reports described in the "Partition Statistics Reports" section list the number of unconnected input and output ports for each partition.
- Do not use tri-state signals or bidirectional ports on hierarchical boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you use boundary tri-states in a lower-level block, synthesis pushes the tri-states through the hierarchy to the top level to take advantage of the tri-state drivers on the output pins of the device.
  In an incremental compilation flow, internal tri-states are supported only when all the destination logic is contained in the same partition, in which case Analysis & Synthesis implements the internal tri-state signals using multiplexing logic. For a bidirectional port that feeds a bidirectional pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition.
- Note that logic is not synthesized or optimized across partition boundaries, which means any constant value (for example, a signal set to `GND`) is not propagated across partitions. If a port is supposed to be connected to `VCC` or `GND`, replace the port with `VCC` or `GND` in the module's design. This allows optimizations to take place that could not be performed if `VCC` or `GND` is connected through a port.
- Do not use the same signal to drive multiple ports on a single partition. If the same driving signal feeds multiple ports of a partition, those ports are logically equivalent. However, because inter-partition optimizations cannot be performed, the compilation of that partition cannot take advantage of this fact, which usually results in sub-optimal performance. For example, if a single clock is used to drive the read and write clocks of a RAM block and the RAM block is compiled separately in a bottom-up design flow, the RAM block is implemented as though there are two unique clocks. If you know the port connectivity will not change (that is, the ports will always be driven by the same signal), redefine the port interface so there is only a single port that can then internally drive other logic in the partition. If required, you can create a wrapper module around the partition that has fewer ports.

- Do not directly connect two ports of a partition. If two ports on a module are directly connected, consider redefining the module to remove those ports. If an output port drives an input port on the same module, 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 by connecting the signals in a higher-level design partition.
- You may have to perform some manual resource balancing across partitions if device resources are overused in the individual partitions. Refer to "Resource Balancing" on page 2–46 for details.
- You may have to perform some timing budgeting if paths that cross partition boundaries require further optimization. Refer to "Timing Budgeting" on page 2–48 for details.

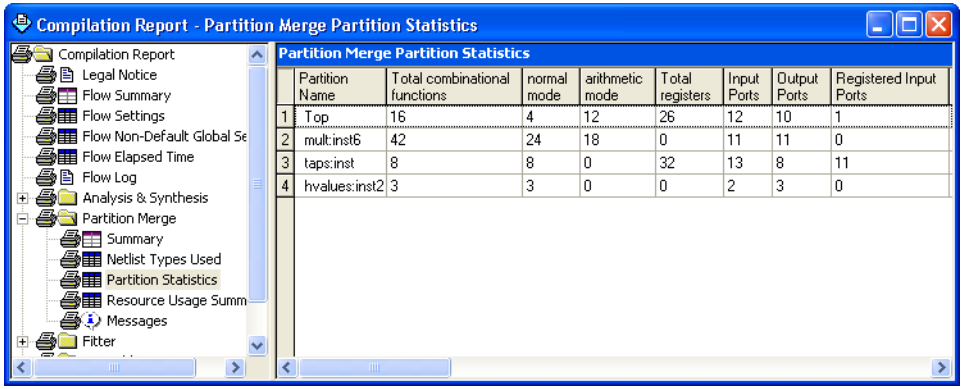
You can use the Incremental Compilation Advisor to check that your design follows many of these guidelines. Refer to "Incremental Compilation Advisor" on page 2–54 for more details.

Partition Statistics Reports

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

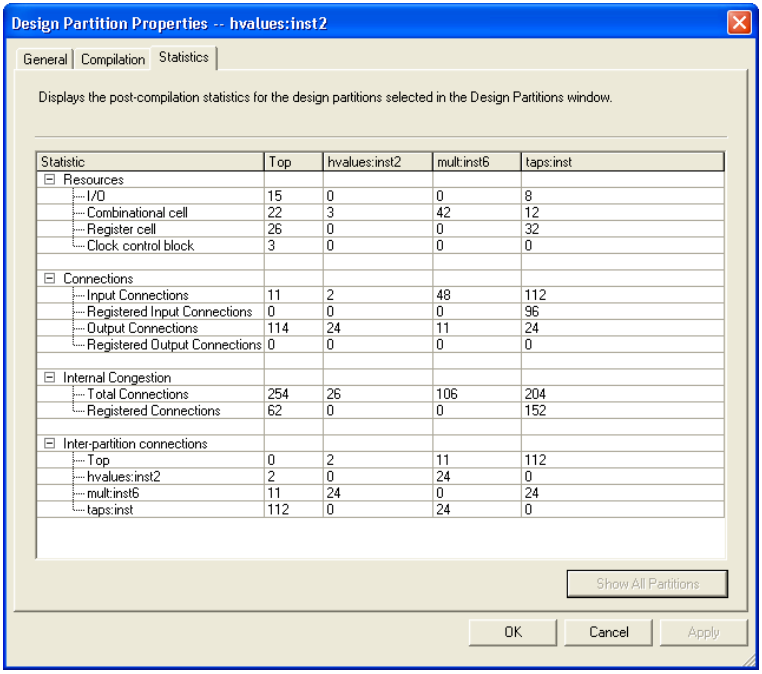
The **Partition Statistics** page under the **Partition Merge** folder of the **Compilation Report** lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells it contains, as well as the number of input and output pins it contains and how many are registered or unconnected. This report is useful when optimizing your design partitions in a top-down compilation flow, or when you are compiling the top-level design in a bottom-up compilation flow, ensuring that the partitions meet the guidelines presented in "Creating Good Design Partitions" on page 2–42. Figure 2–11 shows the report window.

Figure 2–11. Partition Merge Partition Statistics Report



You can also view statistics about the resource and port connections for a particular partition on the **Statistics** tab of the **Design Partition Properties** dialog box. On the Assignments menu, click **Design Partitions Window**. Right-click on a partition and click **Properties** to open the dialog box. Click **Show All Partitions** to view all the partitions in the same report (Figure 2–12).

Figure 2–12. Statistics Tab in the Design Partitions Properties Dialog Box



Resource Balancing

When using incremental compilation, the software synthesizes each partition separately, with no data about the resources used in other partitions. This means that device resources could be overused in the individual partitions during synthesis, and thus the design may not fit in the target device when the partitions are merged.

In a bottom-up design flow in which designers optimize their lower-level designs and export them to a top-level design, the software also places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level.

To avoid these effects, you may have to perform manual resource balancing across partitions.

RAM & DSP Blocks

In the standard synthesis flow, when DSP blocks or RAM blocks are overused, the Quartus II Compiler can perform resource balancing and convert some of the logic into regular logic cells (for example, LEs or ALMs). Without data about resources used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource, such that the design does not fit after all the partitions are merged. In this case, you may be able to manually balance the resources by using 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.

For more information about resource balancing when using Quartus II synthesis, refer to the *Megafunction Inference Control* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For more tips about resource balancing and reducing resource utilization, refer to the appropriate *Resource Utilization Optimization Techniques* section in the *Area & Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Altera recommends using a LogicLock region for each partition to minimize the chance that the logic in more than one partition uses the same logic resource. However, there are situations in which partition placement may still cause conflicts at the top level. For example, you can design a partition one way in a lower-level design (such as using an M-RAM memory block) and then instantiate it in two different ways in the top level (such as one using an M-RAM block and another using an M4K block). In this case, you can use a post-fit netlist only with no placement information to allow the software to refit the logic.

Global Routing Signals

Global routing signals can cause conflicts when multiple projects are imported into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Lower-level partitions can use the same global routing resources, thus causing conflicts at the top level.

In addition, LAB placement depends on whether the inputs to the LCELLs within the LAB are using a global clock signal. Therefore, problems can occur if a design does not use a global signal in the lower-level design, but does use a global signal in the top-level design.

To avoid these problems, the project lead should first determine which partitions will use particular global routing signals. Each designer of a lower-level partition can then assign the appropriate global signals manually, and prevent other signals from using global routing resources. Use the **Global Signal** assignment set to a value of **On** or **Off** in the Assignment Editor to place a signal on a global routing line, or to prevent the signal from using a global routing line. If you want to disable the automatic global promotion performed in the Fitter, turn off the **Auto Global Clock** and **Auto Global Register Control Signals** options. On the Assignments menu, click **Settings**. On the **Fitter Settings** page, click **More Settings** and change the settings to **Off**.

Alternatively, to avoid problems when importing, direct the Fitter to discard the placement and routing of the imported netlist by setting the Fitter preservation level property of the partition to **Netlist Only**. With this option, the Fitter re-assigns all the global signals for this particular partition when compiling the top-level design.

If you are performing a bottom-up flow using the design partition scripts, the software can automatically write the commands to pass global constraints and turn off the automatic options. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–35 for details.

Timing Budgeting

If you optimize lower-level partitions and import them to the top level, any unregistered paths that cross between partitions are not optimized as an entire path. One way to reduce this effect is to ensure input and output ports of the partitions are registered whenever possible.

To ensure that the Compiler correctly optimizes the input and output logic in each partition, you may be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Timing budgets may be required for these I/O ports because when the Compiler optimizes each partition, it has no information about the placement of the logic that connects to that port. If the logic in one partition is placed far away from logic in another partition, the routing delay between the logic could lead to problems meeting the timing requirements. Assigning a timing budget for each part of the connection ensures that the Compiler optimizes the paths appropriately.

When performing manual timing budgeting, you can also use **Virtual Pin** assignments. By assigning location and timing constraints to the **Virtual Pins**, you can further improve the quality of the timing budget.

If you are performing a bottom-up flow using the design partition scripts, the software can write virtual pin assignments and I/O timing budget constraints automatically. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–35 for details.

Methodology to Check Partition Quality During Partition Planning

There is an inherent tradeoff between compilation time and quality of results when you vary the number of partitions in a project. You can ensure that you limit this effect by following an iterative methodology during the partitioning process. In any incremental compilation flow in which you can compile the source code for each partition during the partition planning phase, Altera recommends the following iterative flow:

1. Start with a complete design that is not partitioned and has no location or LogicLock assignments.

2. To perform a placement and timing analysis estimate, on the Processing menu, point to Start and click **Start Early Timing Estimate**.

   You must perform Analysis & Synthesis before performing an Early Timing Estimate. If incremental compilation is already turned on, you must also perform Partition Merge.

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

3. Record the quality of results from the Compilation Report (f_{MAX}, area, and so forth).

4. Create design partitions (and, if required, turn on full incremental compilation) as described in "Preparing a Design for Incremental Compilation" on page 2–10 using the guidelines in "Creating Good Design Partitions" on page 2–42.

5. Perform another Early Timing Estimate or full compilation.

6. Record the quality of results from the Compilation Report. If the quality of results is significantly worse than that obtained in the previous compilation in Step 3, repeat Step 4 through this step (Step 6) to change your partition assignments and use a different partitioning scheme.

7. Even if the quality of results is acceptable, you can repeat Step 4 through Step 6 by further dividing a large partition into several smaller partitions. Doing so improves compilation time in future incremental compilations. You can repeat this step until you achieve a good tradeoff 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).

The Importance of Floorplan Location Assignments in Incremental Compilation

Floorplan location planning is very important for a design that uses full incremental compilation, for two reasons:

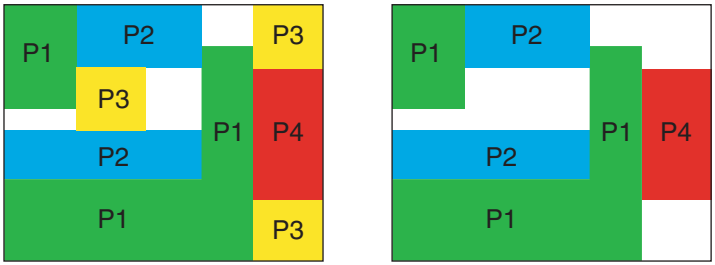
- To avoid resource conflicts between partitions
- To ensure a good quality of results when recompiling partitions and other partition placement is unchanged

Location assignments for each partition ensures that there are no conflicts for locations between different partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating, it is unclear which resources on the device are allocated for the logic associated with the region. Without clearly defining this resource budget, bottom-up design can produce many resource conflicts when importing results, because each bottom-up partition has no information about its resource budget and may therefore claim resources required by another partition.

In addition, a design floorplan helps to avoid the situation that arises when the Fitter is directed to place or replace a portion of the design in an area of the device where most resources have already been claimed. In this case, the placement of the post-fit netlists of other modules forces the Fitter to place the new portion of the design in the empty parts of the device. There are two immediate disadvantages to this situation. First, the Fitter must work harder because of the higher number of physical constraints, and therefore compilation time probably increases. Second, the quality of results often decreases, sometimes dramatically, because the placement of the target module is now scattered throughout the device.

Figures 2–13 and 2–14 illustrate the problems associated with refitting designs that do not have floorplan location assignments. Figure 2–13 shows the initial placement of a four-partition design (**P1**–**P4**) without floorplan location assignments. The second part of the figure shows the situation if a change occurs to **P3**. After removing the logic for the changed partition, the Fitter must replace and reroute the new logic for **P3** using the white space shown in the figure.

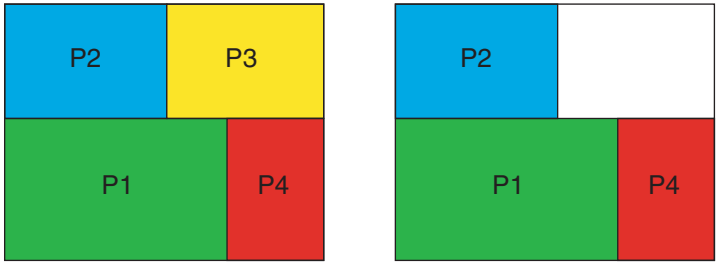
Figure 2–13. Representation of Device Floorplan without Location Assignments



Performing this placement is very difficult. The Fitter may not be able to find any legal placement for the logic in partition **P3**, even if it was able to do so in the initial compilation. If the Fitter does find a legal placement, the results are probably sub-optimal.

Figure 2–14 shows the initial placement of a four-partition design with floorplan location assignments made by the user, and the situation after partition **P3** is removed in this case.

Figure 2–14. Representation of Device Floorplan with Location Assignments



This placement presents a much more reasonable task to the Fitter and yields better results than the previous case that does not have floorplan location assignments.

Creating Good Floorplan Location Assignments

This section presents recommendations for creating a design floorplan using LogicLock regions.

Each LogicLock region should contain logic from only one partition. This organization helps prevent resource conflicts in a bottom-up design and can lead to better performance preservation when locking down parts of a project in a top-down design. One exception to this rule is the case where you want to have two lower-level partitions compiled together in the same LogicLock region because of tight interaction, but you want to separate the placement of the parent logic for each partition. In this case, you can place two partitions in one LogicLock region, but you must ensure that you recompile both partitions every time the logic in one partition changes.

If your design contains hierarchical partitions (that is, parent-child relationships between partitions), you can create hierarchical LogicLock regions to ensure that the logic in the child partition is physically placed inside the LogicLock region for the parent partition. This can be useful when the parent partition does not contain registers at the boundary with the lower-level child partition. To create a hierarchical relationship between regions in the **LogicLock Regions** window, drag and drop the child region to the parent region.

Ensure that all LogicLock regions in the design have a fixed size and have their origin locked to a specific location on the chip. If you use autosized, floating-location regions to create an initial floorplan, be sure to back-annotate the origin and lock the size of the regions. Do not use soft LogicLock regions. Refer to "The Importance of Floorplan Location Assignments in Incremental Compilation" on page 2–50 for more information.

If resource utilization is low, you can enlarge regions chosen by the Fitter with autosizing. Doing so usually improves the final results because it gives the Fitter more freedom to place additional logic added to the partition during future incremental compilations.

Ideally, almost the entire device should be covered by LogicLock regions. Give more area to regions that are densely populated, because overly congested regions can lead to poor results. You may move the region origins from auto-floating region placement to satisfy this requirement, but Altera recommends preserving the Fitter-determined relative placement of the regions. Also, regions that are too large for their logic can result in wasted resources and also lead to poor results. You should define LogicLock regions that are neither too small nor too big.

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 will lead to resource conflicts when importing bottom-up results into a final top-level design.

If two LogicLock regions have several connections between them, place them near each other to improve timing performance. By placing connected regions near each other, the Fitter has more opportunity to optimize inter-region paths when both partitions are recompiled. Reducing the criticality of inter-region paths also allows the Fitter more flexibility when placing the other logic in each region.

You can use the Incremental Compilation Advisor to check that your design follows many of these guidelines. Refer to "Incremental Compilation Advisor" on page 2–54 for more details.

For more information about making and editing LogicLock regions, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

Excluding Certain Device Elements (Such as RAM or DSP Blocks) with Resource Exceptions

If your design contains memory or digital signal processing (DSP) elements, you may want to exclude these elements from the LogicLock region. You can use LogicLock resource exceptions to prevent elements of certain types from being assigned to a region. Note that the filter does not prevent them from being placed inside the region boundaries unless the region's **Reserved** property is turned on. Defining a resource exception instructs the Fitter that certain blocks are not required to be inside a region.

Resource exceptions are useful in cases where it is difficult to place rectangular regions for design blocks that contain memory and DSP elements, because of their placement in columns throughout the device floorplan. Excluding these 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. If desired, you can also create separate regions for the memory or DSP blocks, which can be shaped to accommodate the columns in the device to control the placement of those design elements.

To view any resource exceptions, right-click in the LogicLock Regions window and click **Properties**. In the **LogicLock Region Properties** dialog box, view the **Exceptions** column in the **Members** box. To set up a resource exception, highlight the appropriate region member and click **Edit Exceptions**, then turn on the design element types to be excluded

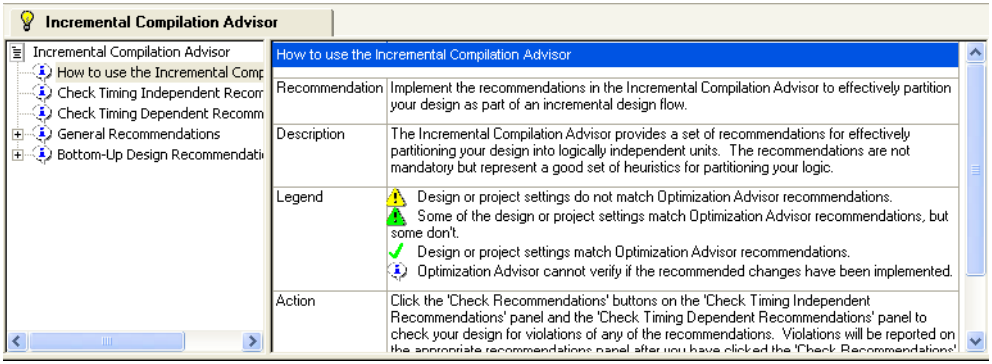
from the region. You can choose to exclude combinational logic or registers from logic cells, or any of the sizes of TriMatrix™ memory blocks, or DSP blocks.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows many of the recommendations presented in this chapter for creating design partitions and floorplan location assignments. On the Tools menu, point to Advisors, and click **Incremental Compilation Advisor**.

As shown in Figure 2–15, recommendations are split into **General Recommendations** that apply to all compilation flows and **Bottom-Up Design Recommendations** that apply to bottom-up design methodologies. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change. 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.

Figure 2–15. Incremental Compilation Advisor



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

The Quartus II software does not currently support Timing Dependent Recommendations for the TimeQuest timing analyzer.

For some recommendations, if your design does not follow the recommendation, the check operation adds a list of the parts of the design that could be improved. For example, if not all of the partition I/O ports follow the **Register All Ports** recommendation, the advisor displays a list of unregistered ports with the partition name and the source and destination nodes for the port.

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

The first time you open the RTL or Technology Map Viewer, a preprocessor stage runs. This preprocessor resets the Incremental Compilation Advisor, so you must rerun the Check Recommendations process. Alternatively, you can open the appropriate netlist viewer before you use the Incremental Compilation Advisor if you want to locate nodes in the viewer.

Criteria for Successful Partition & Floorplan Schemes

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, and consider using another scheme if any of the following guidelines are not met:

- No degradation in f_{MAX} should be observed after the design is partitioned and floorplan location assignments are created. In many cases, a slight increase in f_{MAX} is possible.
- The area increase should be no more than 5% after the design is partitioned and floorplan location assignments are created.
- The time spent in the routing stage should not significantly increase.

The amount of compilation time spent in the routing stage is reported in the Messages window with an Info message indicating 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.

To help you modify your LogicLock regions, you can identify areas of congested routing in your design using the Timing Closure Floorplan. On the Assignments menu, click **Timing Closure Floorplan** and turn on **Show Routing Congestion**. This feature is available only when you click **Field View** on the View menu.

For details about using the Timing Closure Floorplan, refer to the *Timing Closure Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

User Scenarios—Incremental Compilation Application Examples

To better illustrate the applications and behavior of the full incremental compilation flow, this section presents several possible user scenarios. These scenarios are divided into two sections:

- "Top-Down Incremental Design Flow Scenarios"
- "Bottom-Up Incremental Design Flow Scenarios"

Top-Down Incremental Design Flow Scenarios

There are four top-down incremental design flow examples:

- "Scenario 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow"
- "Scenario 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow"
- "Scenario 3—Preserving One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow"
- "Scenario 4—Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow"

All scenarios assume you have set up the project to use the full incremental compilation flow, using the steps described in "Preparing a Design for Incremental Compilation" on page 2–10.

Scenario 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow

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

Perform the following steps to update the single source file:

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

Scenario 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow

Background: You have just performed a lengthy full compilation of a design that consists of multiple partitions. The Timing Analyzer reports that the f_{MAX} timing requirement is not met. After some analysis, you believe that timing closure can be achieved if placement can be improved for one particular partition. You have at least three optimization techniques in mind: 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 (or just one of them) to only the partition in question.

Perform the following steps to raise the Placement Effort Multiplier or enable Physical Synthesis:

1. On the Assignments menu, click **Design Partitions Window**.
2. For the partition in question, set the **Netlist Type** to **Post-Synthesis**. This causes the partition to be placed and routed with the new Fitter settings (but not resynthesized) during the next compilation.

3. For the remaining partitions (including the top-level entity), set the **Netlist Type** to **Post-Fit**. Set the **Fitter Preservation Level** to **Placement** to allow for the most flexibility during routing. To reduce compilation time further, use the **Placement and Routing** setting. These partitions are preserved during the next compilation.

4. Apply the desired optimization settings.

5. Click **Start Compilation** to incrementally compile the design with the new settings. During this compilation, the Partition Merge stage automatically merges the post-synthesis netlist of the critical partition with the post-fit netlists of the remaining partitions. This "merged" netlist is fed to the Fitter. The Fitter then refits only one partition. Since the effort is reduced as compared to the initial full compilation, the compilation time is also reduced.

To use Design Space Explorer, perform the following steps:

1. Repeat steps 1–3 of the previous set of steps.

2. Save the project and run Design Space Explorer.

Scenario 3—Preserving One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow

Background: Prior to any compilation, you have some insight into which partition will be the most critical (in terms of timing) after placement and routing. To help achieve timing closure, you decide to use the following compilation flow.

The critical partition is placed and routed by itself, with all optimizations turned on (manually or through 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.

This flow generally works only if the critical path is contained inside the partition in question. This is one reason why both the inputs and outputs of each partition should be registered.

For this scenario, perform the following steps:

1. Perform partitioning and floorplan location assignment creation.

2. For the partition expected to be critical, on the Assignments menu, click **Design Partitions Window** and set **Netlist Type** to **Source File**.

3. For the remaining partitions (other than any direct or indirect parents of the critical one), set the **Netlist Type** to **Empty**.

4. Click **Start Compilation** to compile with the desired optimizations turned on, or use Design Space Explorer.

5. Check the Timing Analyzer report to ensure that the 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 critical partition. Set the **Fitter Preservation Level** to **Placement and Routing** to preserve the results.

7. Change the **Netlist Type** from **Empty** to **Source File** for the remaining partitions.

8. Turn off the optimizations set in step 4, and compile the design. Turning off the optimizations at this point does not affect the fitted partition, because its Netlist Type is set to **Post-Fit**.

9. Check the Timing Analyzer report to ensure that the timing requirements are met. If not, make design or option changes and repeat step 8 and step 9 until the requirements are met.

This flow is similar to a bottom-up design flow in which a module is implemented separately and is merged into the rest of the design afterwards. Refer to *"Empty Partitions" on page 2–23* for more information about potential issues. Ensure that if there are any partitions representing a design file that is missing from the project, you create a placeholder wrapper file that defines the port interface.

Scenario 4—Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow

Background: Prior to any compilation, you have some insight into which partition will be the most critical (in terms of timing) after placement and routing. To help achieve timing closure, you decide to use the following compilation flow.

Only the non-critical partitions are placed and routed initially, using floorplan location assignments. These non-critical partitions are then preserved when the critical partition is introduced into the Fitter, with various optimizations turned on (manually or through Design Space Explorer).

For this scenario, perform the following steps:

1. Perform partitioning and floorplan creation.
2. For the partition expected to be critical, on the Assignments menu, click **Design Partitions Window** and set the **Netlist Type** to **Empty**.
3. For the remaining partitions, set the **Netlist Type** to **Source File**.
4. Click **Start Compilation** to compile the non-critical partitions.
5. Check the Timing Analyzer report to ensure that the timing requirements are met. If so, proceed to step 6. Otherwise, make design or option changes and 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 processed partitions. Set the **Fitter Preservation Level** to **Placement** to allow for the most flexibility during routing.
7. Change the **Netlist Type** from **Empty** to **Source File** for the partition expected to be critical.
8. Click **Start Compilation** to compile the design with optimizations turned on, or use Design Space Explorer.
9. Check the Timing Analyzer report to ensure that the timing requirements are met. If not, make design or option changes and repeat steps 8 and 9 until the requirements are met.

This flow is similar to a bottom-up design flow, in which a module is implemented separately and merged into the rest of the design afterwards. Refer to *"Empty Partitions" on page 2–23* for more information about potential issues. If there are any partitions representing a design file that is missing from the project, ensure that you create a placeholder wrapper file that defines the port interface.

Bottom-Up Incremental Design Flow Scenarios

There are two bottom-up design flow examples:

- "Scenario 5—Team-Based Bottom-Up Design Flow"
- "Scenario 6—Design Iteration in a Bottom-Up Design Flow"

Scenario 5—Team-Based Bottom-Up Design Flow

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

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

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

1. Create a new Quartus II project that will ultimately contain the full implementation of the entire design.

2. To prepare for the bottom-up methodology, create a "skeleton" of the design that defines the hierarchy for the subdesigns that will be implemented by separate designers. The top-level design implements the top-level entity in the design and instantiates wrapper files that represent each subdesign by defining only the port interfaces but not the implementation.

3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.

4. Ensure that **Full incremental compilation** is turned on.

5. Make design partition assignments for each subdesign and set the Netlist Type for each design partition that will be imported to **Empty** in the Design Partitions window.

6. Create LogicLock regions for each of the lower-level partitions to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications.

7. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command prompt.

8. Make any changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock region, for all partitions and virtual pin location assignments. Altera further recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition to help timing closure during integration at the top level. If lower-level projects have not already been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles.

9. Provide each lower-level designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.

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

If you are using makefiles, perform the following steps:

1. Use the `make` command and the makefile provided by the project lead to create a Quartus II project with all design constraints, and compile the project.

2. The information about which source file should be associated with which partition is not available to the software automatically, so you must specify this information in the makefile. You must specify the dependencies before the software will rebuild the project after the initial call to the makefile.

3. When you have achieved the desired compilation results and the design is ready to be imported into the top-level design, the project lead can use the `master_makefile` command to export this lower-level partition and create a Quartus II Exported Partition file, and then import it into the top-level design.

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

1. Create a new Quartus II project for the subdesign.

2. Make LogicLock region assignments and global assignments (including clock settings) as specified by the project lead.

3. Make Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level module.

4. Make floorplan location assignments to the Virtual Pins so that they are placed in their corresponding regions as determined by the top-level module. This provides the Fitter with more information about the timing constraints between modules. Alternatively, you can apply timing I/O constraints to the paths that connect to virtual pins.

5. Ensure that **Full incremental compilation** is turned on and proceed to compile and optimize the design as needed.

6. When you have achieved the design compilation results, on the Project menu, click **Export Design Partition**. The **Export Design Partition** dialog box appears.

7. Under **Netlist to export**, select the netlist type **Post-fit netlist** to preserve the placement and performance of the subdesign, and turn on **Export routing** to include the routing information if required. You can export **Post-synthesis netlist** instead if placement or performance preservation is not required.

8. Provide the Quartus II Exported Partition file to the project lead.

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

If you are using makefiles, perform the following steps:

1. Use the `master_makefile` command to export each lower-level partition and create Quartus II Exported Partition files, and then import them into the top-level design.

2. The software does not have all the information about which source files should be associated with which partition, so you must specify this information in the makefile. The software cannot rebuild the project if source files change unless you specify the dependencies.

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

1. After you obtain the Quartus II Exported Partition file for each subdesign from the other designers on the team, on the Project menu, click **Import Design Partition** and specify the partition in the top-level project that is represented by the subdesign Quartus II Exported Partition file.

2. Repeat the import process described in step 1 for each partition in the design. After you have imported each partition once, select all the design partitions and use the **Reimport using latest import files at previous locations** option to import all of the files from their previous locations at one time.

Resolving Assignment Conflicts During Import

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

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

When LogicLock region assignment conflicts occur, the project lead may want to do one of the following:

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

The project lead can address all of these situations using the **Advanced Import Settings** as described in **"Importing Assignments & Advanced Import Settings" on page 2–32**.

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

Importing a Partition to be Instantiated Multiple Times

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

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

If you choose to 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.

Scenario 6—Design Iteration in a Bottom-Up Design Flow

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

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

For this scenario, perform the following steps:

1. In the top-level design, on the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command line.

2. Because lower-level projects have already been created for each partition, turn off **Create lower-level project if one does not exist**.

3. Make any additional changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera also recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition.

4. The Quartus II software generates Tcl scripts for all partitions, but in this scenario, you would focus on the partitions that make up the cross-partition critical paths. Following are the important assignments in the script:

    - Virtual pin assignments for module pins not connected to device I/O ports in the top-level design.
    - Location constraints for the virtual pins that reflect the initial top-level placement of the pin's source or destination. These help make the lower-level placement "aware" of its

surroundings in the top-level, leading to a greater chance of timing closure during integration at the top-level.
- INPUT_MAX_DELAY and OUTPUT_MAX_DELAY timing constraints on the paths to and from the I/O pins of the partition. These constrain the pins to optimize the timing paths to and from the pins.

5. The project lead provides the scripts to the low-level designers who source the file.
   - 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 command line, type the following command:

     `quartus_cdb -t <filename>.tcl` ↵
6. The lower-level designers recompile their designs with the new assignments.
7. The lower-level designers re-export their results.
8. The top-level designer re-imports the results.
9. You can now analyze the design to determine if the timing requirements have been achieved. Since the lower-level partitions were compiled with more information about connectivity at the top-level, it is more likely that the inter-partition paths have improved placement which helps to meet the timing requirements.

Incremental Compilation Restrictions

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

The following restrictions and limitations are covered:

- "Using Incremental Compilation with Quartus II Archive Files"
- "OpenCore Plus MegaCore Functions"
- "SignalProbe Pins and Engineering Change Management with the Chip Planner"

- "SignalTap II Logic Analyzer & Logic Analyzer Interface in Bottom-Up Compilation Flows"
- "HardCopy Compilation Flows"
- "Restrictions on Megafunction Partitions"
- "Routing Preservation in Bottom-Up Compilation Flows"
- "Bottom-Up Design Partition Script Limitations"
- "Register Packing & Partition Boundaries"
- "I/O Register Packing"

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. When you archive a project, the database information is not included in the archive unless you include the database files in the Quartus II Archive file (**.qar**). In addition, when you import a design partition into a top-level design, the lower-level design netlist is stored in the project database for the top-level design (the top-level project does not use the original source files or the Quartus II Exported Partition file). If you archive the top-level project, the imported design information is not included unless the database files are included in the Quartus II Archive file.

Altera recommends that you turn on **Include database from compilation and simulation** in the **Archive Project** dialog box if any form of incremental compilation is used so that compilation results are preserved.

OpenCore Plus MegaCore Functions

The circuitry that provides OpenCore Plus MegaCore® functions is currently incompatible with user-defined design partitions.

SignalProbe Pins and Engineering Change Management with the Chip Planner

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

To preserve SignalProbe pins or ECO changes, the partitions must be set to a Netlist Type of Post-fit with the Fitter Preservation Level set to Placement and routing. If any partitions with SignalProbe pins or ECO changes are set to post-fit without routing or to netlist only, the software

issues a warning and internally uses the post-fit netlist with placement and routing. If the partitions are set to use the source code or a post-synthesis netlist, the software issues a warning and the post-fit SignalProbe pins or ECO changes are not included in the new compilation.

If ECO changes affect more than one partition or the connection between any partitions, the partitions become linked and all of the higher-level "parent" partitions up to their nearest common parent are linked and required to have the same netlist type. In this case, the connection between the partitions is actually defined outside of the two partitions immediately affected, so all the partitions must be compiled together. This can cause the recompilation of one partition to force the recompilation of other partitions.

When a SignalProbe pin is created, any pipeline registers are created in the same partition as the node being probed. The SignalProbe output pin is assigned to the top level partition. There is a new connection formed between the top-level partition and the lower-level partition that is being probed. Because of this connection, the lower-level partitions and all of the higher-level "parent" partitions up to the top level become linked and must be compiled together. This can cause the recompilation of one partition to force the recompilation of other partitions.

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

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

SignalTap II Logic Analyzer & Logic Analyzer Interface in Bottom-Up Compilation Flows

The SignalTap® II logic analyzer and External Logic Analyzer Interface are not supported in lower-level projects in a bottom-up incremental compilation flow. These features are supported for the top-level project only after lower-level partitions have been imported, and are fully supported in top-down incremental compilation.

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*.

HardCopy Compilation Flows

Quartus II incremental compilation is not supported for HardCopy APEX or HardCopy Stratix design flows.

Top-down incremental compilation is supported for HardCopy II for both the Stratix II first and HardCopy II first flows. Design partitions are migrated to the companion device. LogicLock regions are suggested for design partitions but are not migrated to the companion device, due to the different device architecture. The first compilation after a device migration is a full compilation because of the different device architecture, but subsequent compilations can be incremental if changes to the source code are not required. For example, you can make changes to PLL phase settings without recompiling the HardCopy II design.

Bottom-up incremental compilation is not supported in HardCopy II compilations when there is a migration device set (that is, for HardCopy II compilations with a Stratix II migration device, or Stratix II compilations with a HardCopy II migration device). You can use bottom-up incremental compilation for stand-alone HardCopy II compilations.

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

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 the creation of 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 creation of a partition for any Quartus II internal hierarchy that is dynamically generated during compilation to implement the contents of a megafunction.

Routing Preservation in Bottom-Up Compilation Flows

There are some cases in which routing information cannot be preserved exactly, especially in bottom-up compilation, because of legality in the device architecture. For example, when multiple partitions are imported, there may be routing conflicts because you cannot pre-assign routing for each lower-level block. In addition, if an imported LogicLock region is moved in the top-level design, the relative placement of the nodes is preserved but the routing may not be preserved.

Bottom-Up Design Partition Script Limitations

The Quartus II software version 6.1 has some limitations related to bottom-up design partition scripts. Some of these limitations may be removed in future versions of the software.

Wildcard Support in Bottom-Up Design Partition Scripts

When applying constraints with wildcards, 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 & PLLs in Bottom-Up Design Partition Scripts

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

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

In addition, if you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication or phase shift factors in the PLL. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained or constrained with the incorrect frequency.

Virtual Pin Timing Assignments in Bottom-Up Design Partition Scripts

The design partition scripts use INPUT_MAX_DELAY and OUTPUT_MAX_DELAY assignments to specify the 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 the software sets 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, you can also enter negative numbers for input and output delay values.

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

When a single top-level I/O port drives multiple pins on a lower-level module, it unnecessarily restricts the quality of the synthesis and placement at the lower-level. This occurs because in the lower-level design, the software must maintain the hierarchical boundary and cannot use any information about pins being logically equivalent at the top level. In addition, because I/O constraints are passed from the top-level pin to each of the children, it is possible to have more pins in the lower level than at the top level, and these pins use the top-level I/O constraints and placement options that might make them impossible to place at the lower-level. The software avoids this situation when 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 then the connection is split into multiple signals within the lower-level partition.

Support for the TimeQuest Timing Analyzer & SDC Constraints

If you use constraints with the TimeQuest Timing Analyzer, the assignments are not passed to the lower levels. You must use constraints made for the Classic Timing Analyzer. You can then use the **Generate SDC File from QSF** command to convert the Classic Timing Analyzer constraints in a Quartus II Settings File (**.qsf**) to a Synopsys Design Constraints (**.sdc**) for the TimeQuest analyzer.

For more information about timing constraints and conversion from the QSF file to the SDC file, refer to *Classic Timing Analyzer*, *TimeQuest Timing Analyzer*, and *Switching to the TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Register Packing & Partition Boundaries

The Quartus II software automatically performs register packing during compilation. However, when incremental compilation is enabled, logic in different partitions cannot be packed together because partition boundaries prevent cross-boundary optimization. (Refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–41 for more information.) This restriction applies to all types of register packing, including I/O cells, DSP blocks, sequential logic, and unrelated logic.

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 hierarchy level (and the Top partition), but the corresponding I/O registers exist in other partitions.

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

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

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

- The register feeds exactly one output pin
- The output pin is fed by only one signal
- The path between the register and the 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 the tri-state logic are defined in the same partition.

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

The restrictions on tri-state logic are due to the fact that the I/O atom (device primitive) is created as part of the partition that contains the 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 the 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 and the path between the I/O atom and the I/O pin must include only ports of partitions that have one fan-out each.

Examples of I/O Register Packing Across Partition Boundaries

The following examples provide detailed explanations for various I/O and partition configurations. The examples use BDF schematics to illustrate the design logic.

Example 1—Output Register in Partition Feeding Output Pin
In this example, a subdesign contains a single register, as shown in Figure 2–16. As shown in Figure 2–17, 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.

Figure 2–16. Subdesign with One Register, Designated as a Separate Partition

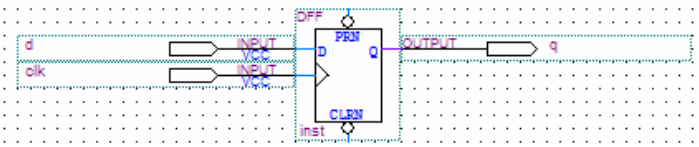
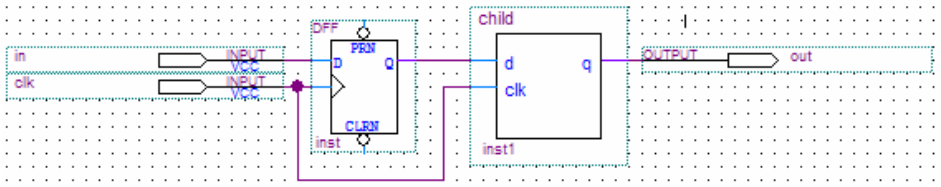


Figure 2–17. Top-level Design Instantiating the Subdesign in Figure 2–16 as an Output Register

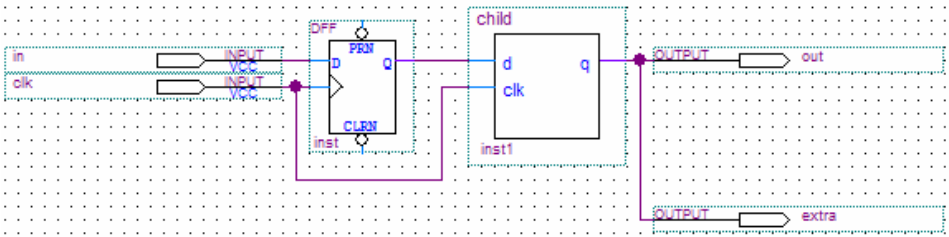


The Quartus II software performs cross-partition register packing if there is a Fast Output Register assignment on pin `out`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed and the partition port feeds an output pin directly.

Example 2—Output Register in Partition Feeding Multiple Output Pins

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–16. The top-level design instantiates the subdesign as an output register with more than one fan-out signal, as shown in Figure 2–18.

Figure 2–18. Top-level Design Instantiating the Subdesign in Figure 2–16 with Two Output Pins



In this case, the software does not perform output register packing. If there is a Fast Output Register assignment on pin `out`, the software issues a warning that the Fitter can't pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This kind of cross-partition register packing is not permitted because it would require modification to the interface of the subdesign partition. In order to perform incremental compilation, the interface of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–16 with the output pin `out` in Figure 2–18, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the output pin. The simplest option is to move the register from the subdesign partition into the partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register feeds only one output pin. Turn off the Analysis & Synthesis setting **Remove Duplicate Registers**. Duplicate the register in your subdesign HDL as in Figure 2–19 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 2–20. This converts the cross-partition register packing into the simplest case where the register has a single fan-out.

Figure 2–19. Modified Subdesign from Figure 2–16 with Two Output Registers & Two Output Ports

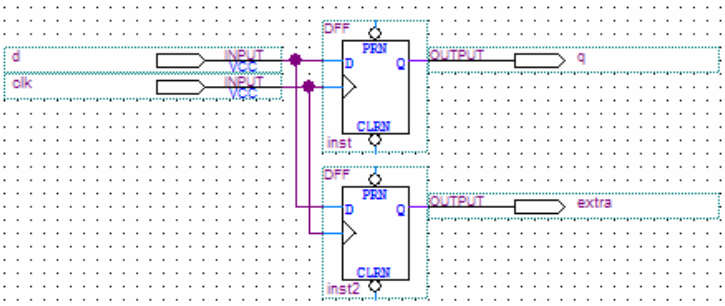
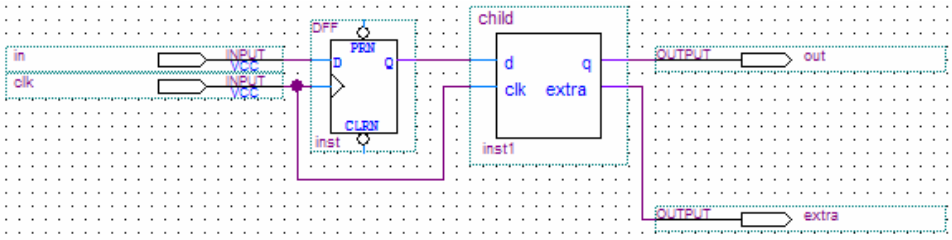


Figure 2–20. Modified Top-Level Design from Figure 2–18 Connecting Two Output Ports to Output Pins



Example 3—Output Register, Output Enable Register & Tri-State Logic in Partition Feeding Output Pin

In this example, a subdesign designated as a separate partition contains an output register, an output enable register, and the tri-state logic to drive the output pin, as shown in Figure 2–21. The top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, as shown in Figure 2–22.

Figure 2–21. Subdesign with Output Register, Output Enable Register & Tri-State Logic, Designated as a Separate Partition

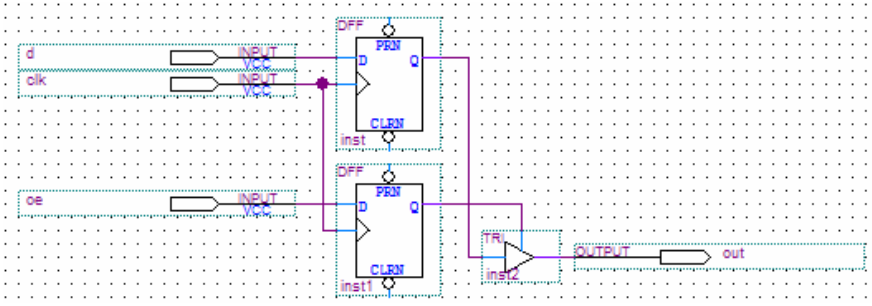
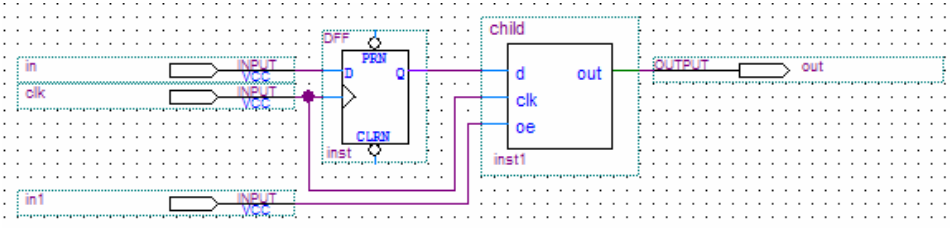


Figure 2–22. Top-level Design Instantiating the Subdesign in Figure 2–21



The Quartus II software performs cross-partition register packing if there is a Fast Output Register assignment, Fast Output Enable Register assignment, or both, on pin `out`. This kind of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed, no logic needs to be optimized across the partition boundary, and the partition port feeds an output pin directly.

Example 4—Output Register, Output Enable Register, or Both, in Partition Feeding Tri-State Output Pin

In this example, a subdesign designated as a separate partition contains two registers, as shown in Figure 2–23. The top-level design instantiates the subdesign with the registers driving the output and the output enable signal for an output pin, as shown in Figure 2–24.

Figure 2–23. Subdesign with Two Registers, Designated as a Separate Partition

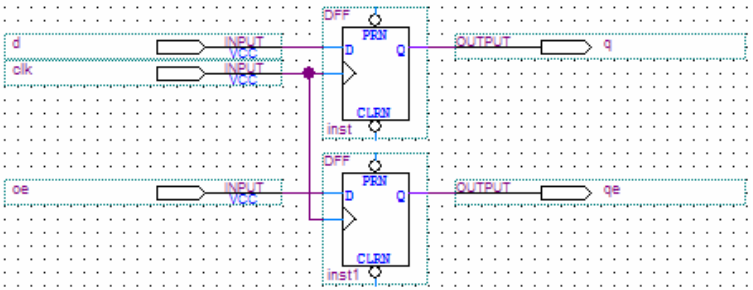
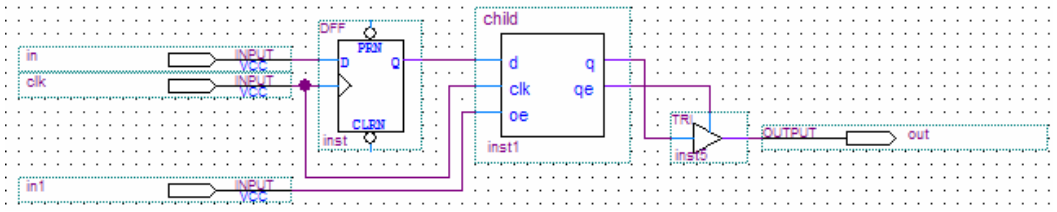


Figure 2–24. Top-level Design Instantiating the Subdesign in Figure 2–25 to Drive Output Enable Logic



In this case, the software cannot perform register packing. If there is a Fast Output Register or Fast Output Enable 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.

The same restrictions apply in the case that the top-level design includes either the output register or the output enable register as well as the tri-state logic. The software cannot pack the register that is part of the subdesign partition into the I/O register.

This type of register packing is not permitted because it would require moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the registers must be moved out of the subdesign partition or the tri-state logic must be moved into the subdesign partition. In order to guarantee correctness of the design with subsequent incremental compilations, the contents of design partitions must be preserved.

To allow the software to pack the output register, output enable register, or both, in the subdesign from Figure 2–23 with the output pin `out` in Figure 2–24, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not need to perform cross-boundary optimizations.
- Restructure your HDL code to 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 top-level partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register and the tri-state logic are contained in the same partition. Move the tri-state logic from the top-level block into the subdesign with both registers as shown in Figure 2–21. Then connect the subdesign to an output pin in the top-level design, as shown in Figure 2–22.

Example 5—Bidirectional Logic in Partition Feeding Bidirectional Pin
The behavior for bidirectional pins is similar to that of an output pin with an output enable. To allow register packing, the registers must be included in the same partition as the tri-state logic that drives the bidirectional pin.

In this example, a subdesign designated as a separate partition contains three registers and the tri-state logic for a bidirectional pin, as shown in Figure 2–25. The top-level design instantiates the subdesign with ports feeding bidirectional and output pins, as shown in Figure 2–26.

Figure 2–25. Subdesign with Three Registers & Tri-State Logic, Designated as a Separate Partition

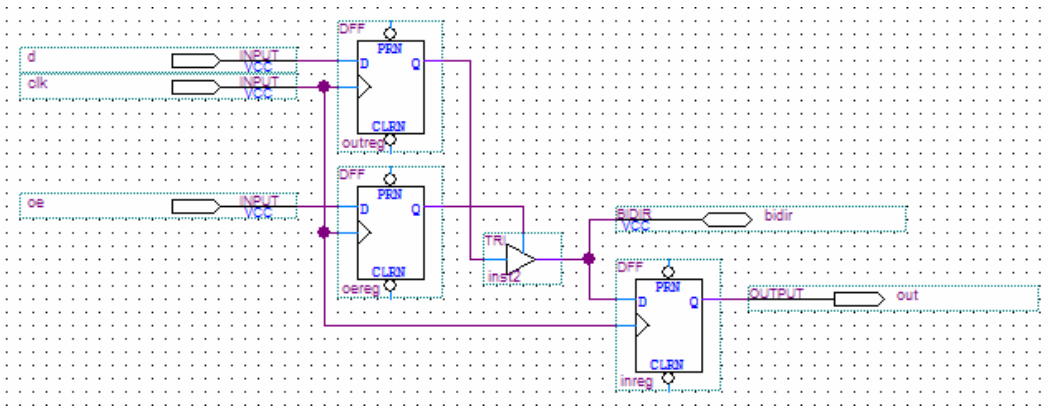
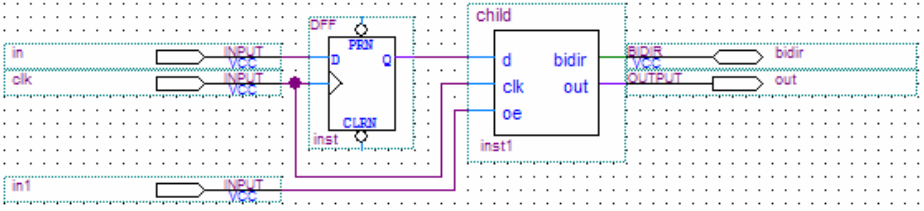


Figure 2–26. Top-level Design Instantiating the Subdesign in Figure 2–28



The Quartus II software performs cross-partition register packing if there is a Fast Output Register, Fast Output Enable Register, or Fast Input Register assignment on pin `bidir`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed and the partition port feeds a bidirectional pin directly.

Registers cannot be packed in designs that have the registers and tri-state logic in different partitions. The situations described in "Example 4—Output Register, Output Enable Register, or Both, in Partition Feeding Tri-State Output Pin" on page 2–77 apply similarly to bidirectional pins if you replace the output pin `out` with a bidirectional pin in the top-level design.

Example 6—Input Register in Partition Fed by Input Pin
In this example, a subdesign contains a single register, as shown in Figure 2–27. The top-level design instantiates the subdesign with a single `fanin` directly fed by an input pin, as shown in Figure 2–28, and designates the subdesign to be a separate design partition.

Figure 2–27. Subdesign with One Register, Designated as a Separate Partition

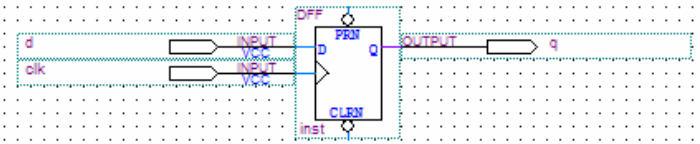
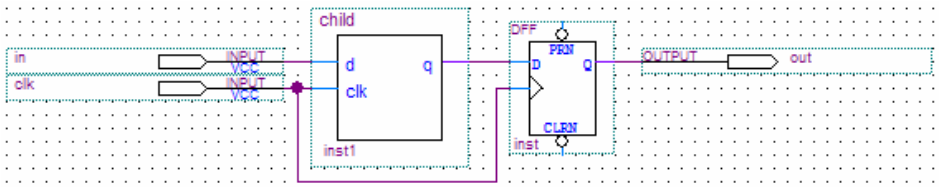


Figure 2–28. Top-level Design Instantiating the Subdesign in Figure 2–27 as an Input Register

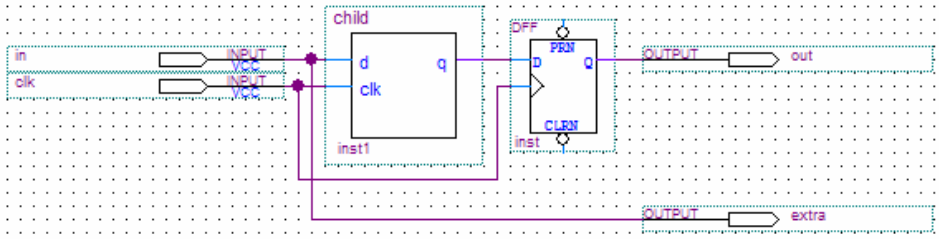


The Quartus II software performs cross-partition register packing if there is a Fast Input Register assignment on pin `in`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not have to be changed and the partition port is fed by an input pin directly.

Example 7—Input Register in Partition Fed by Input with Multiple Fan-Out

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–27. The top-level design instantiates the subdesign as an input register but the input pin also feeds another destination, as shown in Figure 2–29.

Figure 2–29. Top-level Design Instantiating the Subdesign in Figure 2–27 as an Input Register for a Pin with Two Destinations



In this case, the software does not perform input register packing. If there is a Fast Input Register assignment on pin `in`, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This type of cross-partition register packing is not permitted because it would require modification to the interface of the subdesign partition. In order to perform incremental compilation, the interface of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–27 with the input pin `in` in Figure 2–29, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it also prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the input pin. The simplest option is to move the register from the subdesign partition into the partition containing the input pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.

Example 8—Inverted Input Register in Partition Fed by Input Pin
In this example, a subdesign designated as a separate partition contains an inverted register as in Figure 2–30. The top-level design instantiates the subdesign as an input register, as shown in Figure 2–31.

Figure 2–30. Subdesign with an Inverted Register, Designated as a Separate Partition

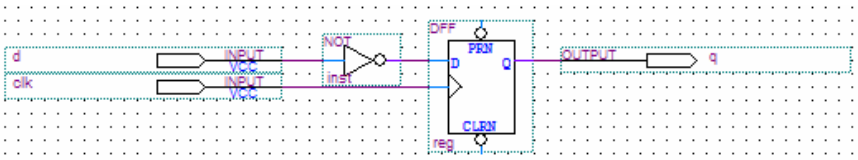
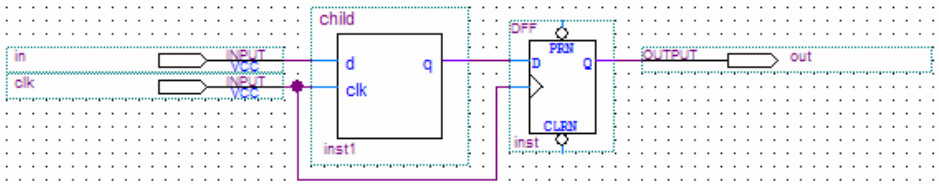


Figure 2–31. Top-level Design Instantiating the Subdesign in Figure 2–30 as an Input Register



The Quartus II software performs cross-partition register packing if there is a Fast Input Register assignment on pin `in`. This kind of cross-partition input register packing is permitted because the software can implement the logic for the inversion with the input register inside the partition, and then the partition port is fed by an input pin directly.

Example 9—Input Register in Partition Fed by Inverted Input Pin, or Output Register in Partition Feeding Inverted Output Pin

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–32. The top-level design in Figure 2–33 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 2–34 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

Figure 2–32. Subdesign with One Register, Designated as a Separate Partition

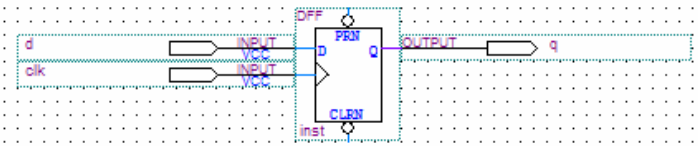


Figure 2–33. Top-level Design Instantiating the Subdesign in Figure 2–32 as an Input Register with an Inverted Input Pin

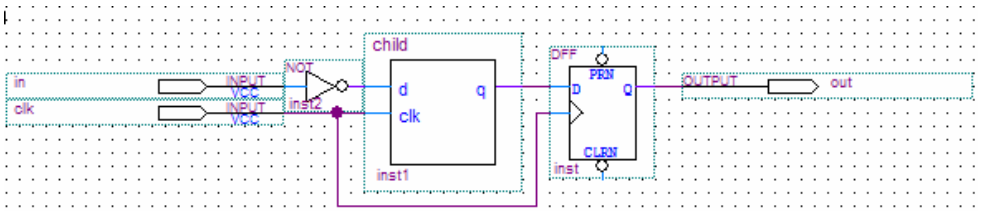
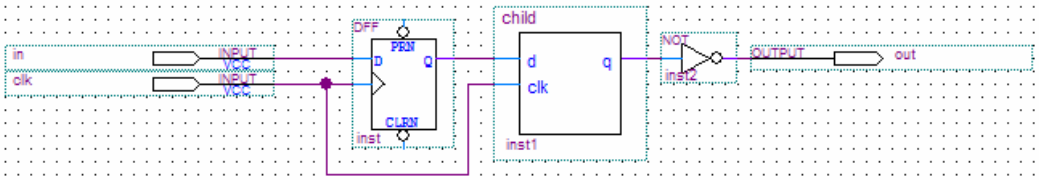


Figure 2–34. Top-level Design Instantiating the Subdesign in Figure 2–33 as an Output Register Feeding an Inverted Output Pin



In these cases, the software does not perform register packing. If there is a Fast Input Register assignment on pin `in` in Figure 2–33 or a Fast Output Register assignment on pin `out` in Figure 2–34, the software issues a warning that the Fitter can't 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 register packing is not permitted because it would require 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. In order to guarantee correctness of the design with subsequent incremental compilations, the contents of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–32 with the input pin `in` in Figure 2–33 or the output pin `out` in Figure 2–34, make one of the following changes:

- Remove the design partition assignment from the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the pin. The simplest option is to move the register from the subdesign partition into the top-level partition containing the pin. This ensures that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register and the inverter are contained in the same partition. Move the inverter from the top-level block into the subdesign as shown in Figure 2–30 for an input pin. Then connect the subdesign to a pin in the top-level design, as shown in Figure 2–31 for an input pin.

Scripting Support

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

```
quartus_sh --qhelp ↵
```

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

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

Generate Incremental Compilation Tcl Script Command

To create a template Tcl script for full incremental compilation, use the Generate Incremental Compilation Tcl Script feature. Right-click in the **Design Partitions Window** and click **Generate Incremental Compilation Tcl Script**.

If you have made any partition assignments in the user interface, this script contains the Tcl equivalents of the assignments. The Tcl assignments are described in the following sections.

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)
- `INCREMENTAL_SYNTHESIS`—Incremental synthesis only
- `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 *Node-Naming Conventions in Quartus II Integrated Synthesis* 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, and you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

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

Setting Properties of Design Partitions

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

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

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

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

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

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

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

- `NETLIST_ONLY`—Netlist Only
- `PLACEMENT`—Placement
- `PLACEMENT_AND_ROUTING`—Placement and Routing

For details about these partition properties, refer to "Setting Properties of Design Partitions".

Creating Good 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 LogicLock 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 following keywords:

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 is used to specify a resource filter for the **alu:alu_unit** entity assigned to the ALU region. The value of the assignment takes the same format as the resource listing string taken by the previous Tcl command.

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

Generating Bottom-Up Design Partition Scripts

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

```
generate_bottom_up_scripts <options> ↵
```

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

```
load_package database_manager
```

You must open a project before you can generate scripts.

The Tcl options are the same as those available in the GUI. The exact format of each option is specified in Table 2–5.

Table 2–5. Options for Generating Bottom-Up Partition Scripts with Tcl Commands

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

Note to Table 2–5:
(1) No default.

The following example shows how to use the Tcl command:

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

Command Line Support

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

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

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

The command prompt options are the same as those available in the GUI. They are listed in Table 2–6.

Table 2–6. Options for Generating Bottom-Up 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 2–6:
(1) No default. You must provide this option if you are including virtual pin timing.

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

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

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

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

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

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

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

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

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

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

```
set_global_assignment -name AUTO_EXPORT_INCREMENTAL_COMPILATION ON
```

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

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

```
quartus_cdb -- INCREMENTAL_COMPILATION_IMPORT ↵
```

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

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

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

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

Make Files

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

User Scenarios—Incremental Compilation 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 \
 db/A_inst -to A -section_id "Partition_A"
set_instance_assignment -name PARTITION_HIERARCHY \
 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
```

Scenario 1—Changing a Source File for One of Multiple Partitions

Background: You have run the initial compilation shown in the example script under "User Scenarios—Incremental Compilation Application Examples" on page 2–92. You have modified the HDL source file for partition **A**, and would like to recompile it.

Run the standard flow compilation command in your Tcl script:

```
execute_flow –full_compile
```

Or, run 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.

Scenario 2—Optimizing the Placement for One of Multiple Partitions

Background: You have run the initial compilation shown in the example script under "User Scenarios—Incremental Compilation Application Examples" on page 2–92. 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 the following script:

```
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 the performance of unchanged logic in your design as you make changes elsewhere. The various applications of incremental compilation enable you to improve your productivity while designing for high-density FPGAs, using either top-down or bottom-up design methodologies. Using the techniques and recommendations presented in this chapter allows you to make good design decisions to achieve timing closure while reducing design iteration time by an average of about 60%.

Document Revision History

Table 2–7 shows the revision history for this document.

Table 2–7. Documentation Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Chapter 2 was formerly Chapter 1 in version 6.0.0. Reorganized chapter to group recommendations and guidelines together. Updated for the Quartus II software version 6.1: • Added support for Stratix III devices. • Added information on the "Incremental Compilation Advisor". • The full incremental compilation option is now turned on by default. • Added new feature for "Exporting a Lower-Level Block within a Project". • Changed the location of the **Automatically export design partition after compilation** option. • Added support for "HardCopy Compilation Flows". • Added that routing can be exported in bottom-up flows. • Added I/O port guidelines in "Creating Good Design Partitions". • Updated limitations: "SignalProbe Pins and Engineering Change Management with the Chip Planner".	Added support for Stratix III devices. Added information about new features and updates in the Quartus II software version 6.1.
May 2006 v6.0.0	Name changed to *Quartus II Incremental Compilation for Hierarchical & Team-Based Design*. Updated for the Quartus II software version 6.0.0 • Added new device support information. • Added top-down and bottom-up design flow information. • Added incremental compilation design compiling information. • Added recommendations for creating good floorplan location assignments. • Added register packing & partition boundary information. • Added engineering management with the Chip Editor. • Added information on how to check and save to reapply SignalProbe. • Added user scenarios.	
December 2005 v5.1.1	Minor typographic update.	

Table 2–7. Documentation Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	
August 2005 v5.0.1	Added documentation on cross-partition register packing.	
May 2005 v5.0.0	Initial release.	

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

QII51002-6.1.0

Introduction

The feature-rich Quartus® II software helps you shorten your design cycles and reduce time-to-market. With FLEX®, ACEX®, APEX™, Stratix® II, Stratix GX, Stratix, Cyclone™ II, Cyclone™, MAX®, and MAX II family support, the Quartus II software is the most widely accepted Altera® design software tool today.

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

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

Chapter Overview

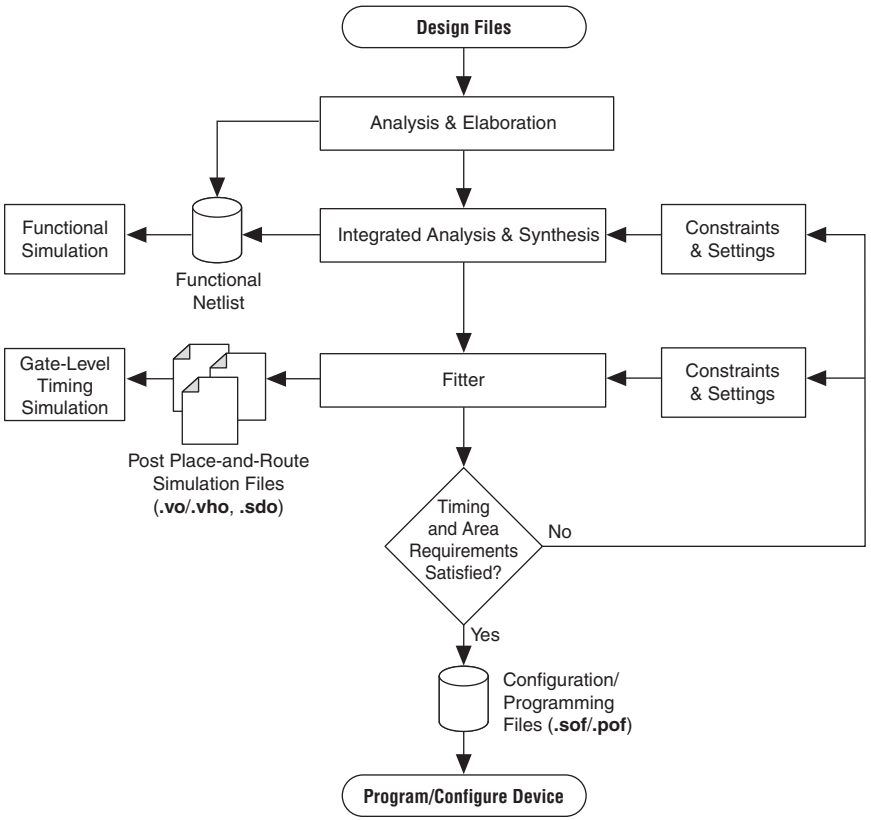
This chapter covers the following topics:

- Typical Design Flow
- Device Support
- Quartus II GUI Overview
- Setting Up MAX+PLUS II Look & Feel in Quartus II
- Compiler Tool
- Quartus II Design Flow
- Quick Menu Reference

Typical Design Flow

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

Figure 3–1. Quartus II Software Design Flow



Device Support

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

Table 3–1. Device Support Comparison

Device Supported	Quartus II	MAX+PLUS II
MAX II	✓	—
Classic™	—	✓
MAX 3000A	✓	✓
MAX 7000S/AE/B	✓	✓
MAX 7000E	—	✓
MAX 9000	—	✓
ACEX® 1K	✓	✓
FLEX® 6000	✓	✓
FLEX 8000	—	✓
FLEX 10K	✓ *(1)*	✓
FLEX 10KA	✓	✓
FLEX 10KE	✓ *(2)*	✓
Mercury™	✓	—
APEX™ 20K/ APEX II	✓	—
Stratix	✓	—
Stratix GX	✓	—
Stratix II	✓	—
Cyclone™	✓	—
Cyclone II	✓	—
Hardcopy® Series	✓	—

Notes to Table 3–1:

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

(2) Some packages are not supported.

Quartus II GUI Overview

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

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

Project Navigator

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

Node Finder

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

Tcl Console

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

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

Messages

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

Status

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

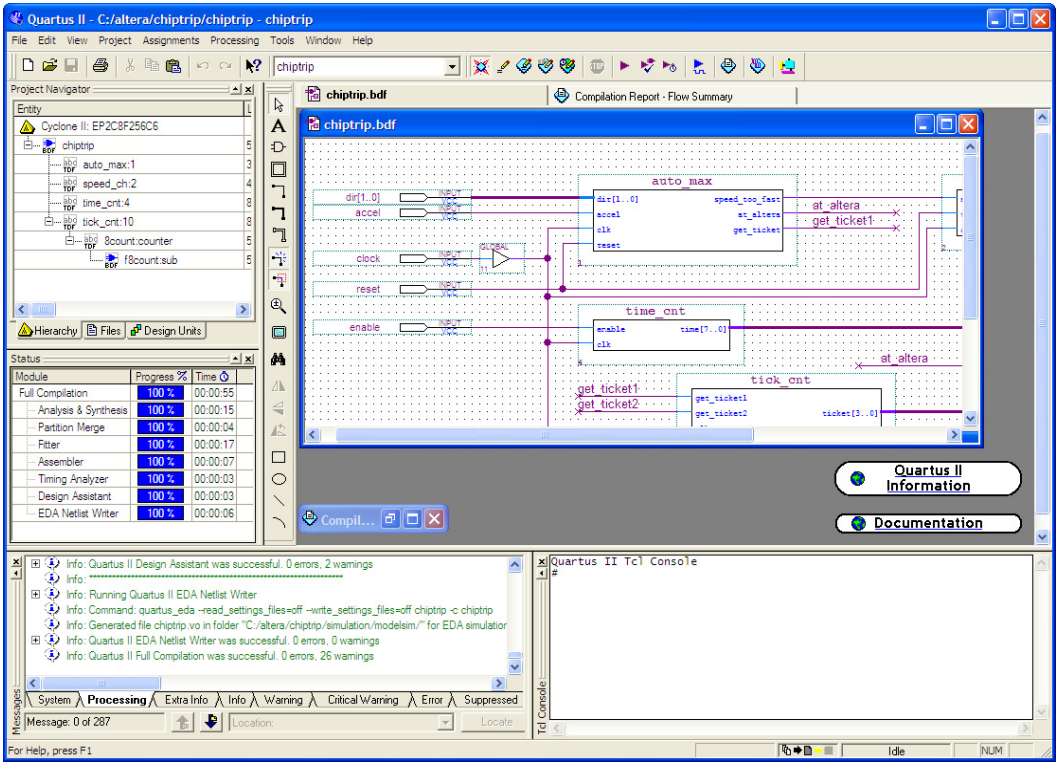
Change Manager

The Change Manager provides detailed tracking information on all design changes made with the Chip Editor.

For more information about the Engineering Change Manager and the Chip Editor, refer to the *Design Analysis & Engineering Change Management with Chip Editor* chapter in volume 3 of the *Quartus II Handbook*.

Figure 3–2 shows a typical Quartus II software display.

Figure 3–2. Quartus II Look & Feel



Setting Up MAX+PLUS II Look & Feel in Quartus II

You can choose the MAX+PLUS II look and feel by selecting MAX+PLUS II in the **Look & Feel** box of the **General** tab of the **Customize** dialog box on the Tools menu.

Any changes to the look and feel do not become effective until you restart the Quartus II software.

By default, when you select the MAX+PLUS II look and feel, the **MAX+PLUS II** quick menu (Figure 3–21 on page 3–35) appears on the left side of the menu bar. You can turn the Quartus II and MAX+PLUS II quick menus on or off. You also can change the preferred positions of the two quick menus. To change these options, perform the following steps:

1. On the Tools menu, click **Customize**. The **Customize** dialog box is shown.
2. Click the **General** tab.
3. Under **Quick menus**, select your preferred options.

MAX+PLUS II Look & Feel

The MAX+PLUS II look and feel in the Quartus II software closely resembles the MAX+PLUS II software. Figures 3–3 and 3–4 compare the MAX+PLUS II software appearance with the Quartus II MAX+PLUS II look and feel.

Figure 3–3. MAX+PLUS II Software GUI

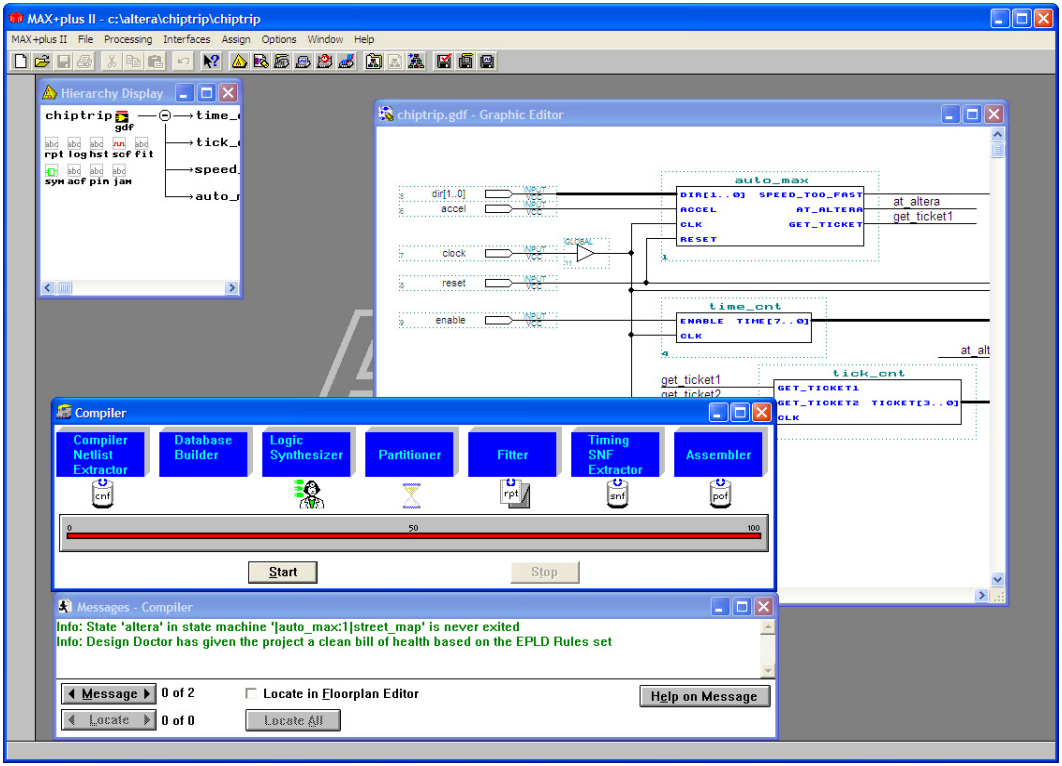
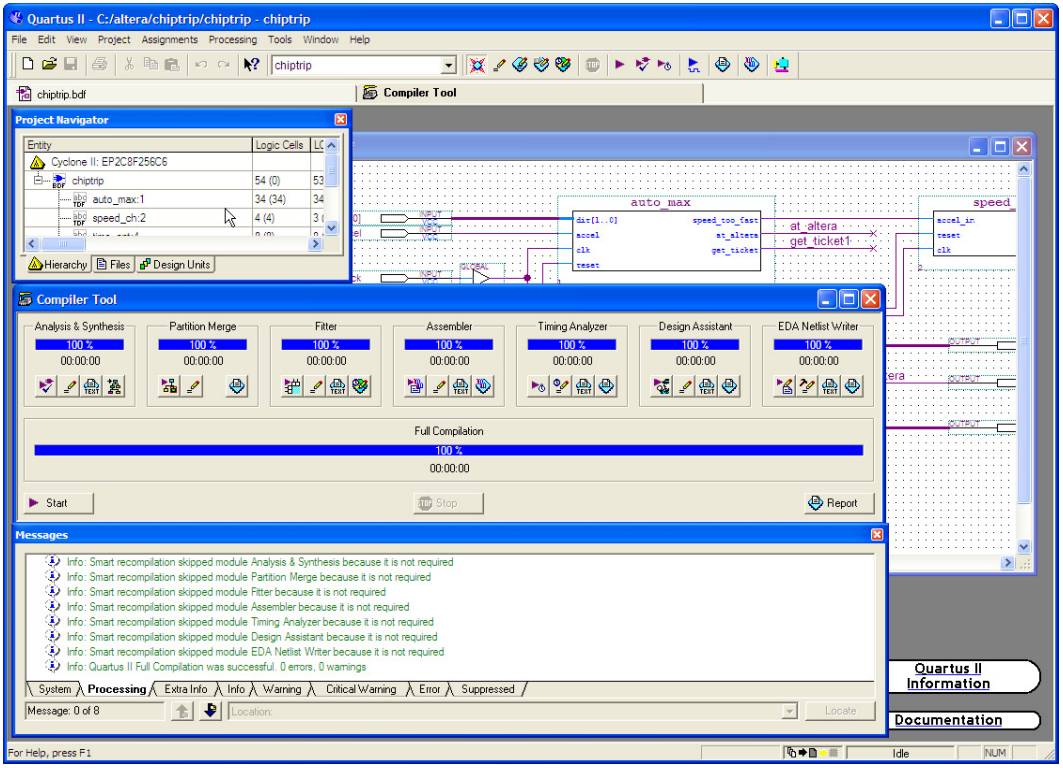


Figure 3–4. Quartus II Software with MAX+PLUS II Look & Feel



The standard MAX+PLUS II toolbar is also available in the Quartus II software with the MAX+PLUS II look and feel in the Quartus II software (Figure 3–5).

Figure 3–5. Standard MAX+PLUS II Toolbar

Compiler Tool

The Quartus II Compiler Tool provides an intuitive MAX+PLUS II style interface. You can edit the settings and view result files for the following modules:

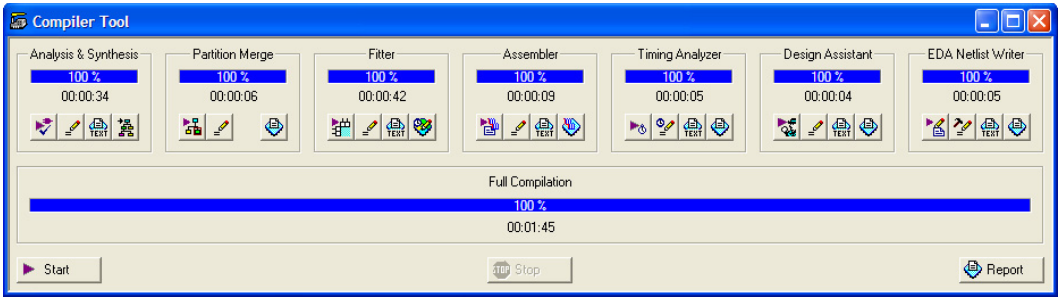
- Analysis & Synthesis
- Partition Merge
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer
- Design Assistant

Each of these modules is described later in this section.

To start a compilation using the Compiler Tool, click **Compiler Tool** from either the MAX+PLUS II menu or the Tools menu and click **Start** in the Compiler Tool. The Compiler Tool, shown in Figure 3–6, displays all modules, including optional modules such as Partition Merge, Assembler, EDA Netlist Writer, and the Design Assistant.

For information about using the Quartus II software modules at the command line, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Figure 3–6. Running a Full Compilation with the Compiler Tool



Analysis & Synthesis

The Quartus II Analysis & Synthesis module analyzes your design, builds the design database, optimizes the design for the targeted architecture, and maps the technology to the design logic.

In MAX+PLUS II software, these functions are performed by the Compiler Netlist Extractor, Database Builder, and Logic Synthesizer. There is no module in the Quartus II software similar to the MAX+PLUS II Partitioner module.

Partition Merge

The optional Quartus II Partition Merge module merges the partitions to create a flattened netlist for further stages of the Quartus II compilation flow. The Partition Merge module is not similar to the MAX+PLUS II Partitioner. This tool is available only if you turn on incremental compilation. You can turn on incremental compilation by performing the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the **+** icon to expand **Compilation Process Settings**, and select **Incremental Compilation**. The **Full Incremental Compilation** page appears.
3. Under **Incremental compilation**, turn on Incremental Compilation.

Fitter

The Quartus II Fitter module uses the PowerFit™ fitter to fit your design into the available resources of the targeted device. The Fitter places and routes the design. The Fitter module is similar to the Fitter stage of the MAX+PLUS II software.

Assembler

The optional Quartus II Assembler module creates a device programming image of your design so that you can configure your device. You can select from the following types of programming images:

- Programmer Object File (**.pof**)
- SRAM Output File (**.sof)**
- Hexadecimal (Intel-Format) Output File (**.hexout**)
- Tabular Text File (**.ttf)**
- Raw Binary File (**.rbf**)
- Jam™ STAPL Byte Code 2.0 File (**.jbc)**
- JEDEC STAPL Format File (**.jam**)

You can turn off the Assembler module during compilation by turning off **Run assembler** in the **Compilation Process Settings** page in the **Settings** dialog box. You also can turn off the Assembler by right-clicking in the Compiler Tool window. The Assembler module is similar to the Assembler stage of the MAX+PLUS II software.

Timing Analyzer

The Quartus II Timing Analyzer allows you to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer. The Quartus II Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains, and also reports both f_{MAX} and slack. Slack is the margin by which the timing requirement is met or is not met. For more information on the Timing Analyzer, refer to "Timing Analysis" on page 3–27.

EDA Netlist Writer

The optional Quartus II EDA Netlist Writer module generates a netlist for simulation with an EDA simulation tool. The EDA Netlist Writer module is comparable to the VHDL and Verilog Netlist Writer in the MAX+PLUS II software.

Design Assistant

The optional Quartus II Design Assistant module checks the reliability of your design based on a set of design rules. The Design Assistant analyzes and generates messages for a design targeting any Altera device and is especially useful for checking the reliability of a design to be converted to HardCopy series devices. The Design Assistant is similar to the Design Doctor in the MAX+PLUS II software.

In the Quartus II software, you can reduce subsequent compilation time significantly by turning **Use Smart compilation** on before compiling your design. The Smart Compilation feature skips any compilation stages which are not required and which may use more disk space. This Quartus II smart compilation option is similar to the MAX+PLUS II **Smart Recompile** command. To turn the **Use Smart compilation** option on, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Turn on **Use Smart compilation**.

MAX+PLUS II Design Conversion

With the Quartus II software, you can open MAX+PLUS II designs and convert MAX+PLUS II assignments and files.

The Quartus II software is project based. All the files for your design (HDL input, simulation vectors, assignments, and other relevant files) are associated with a project file. For more information about creating a new project, refer to "Creating a New Project" on page 3–16.

Converting an Existing MAX+PLUS II Design

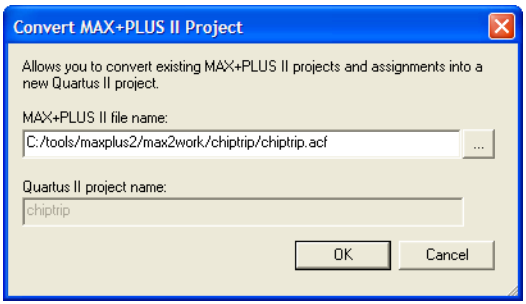
You can easily convert an existing MAX+PLUS II design for use with the Quartus II software with the **Convert MAX+PLUS II Project** command in the Quartus II software or the **Open Project** command. You can find these commands on the File menu

If you use the **Convert MAX+PLUS II Project** command, browse to the MAX+PLUS II Assignments and Configuration File (**.acf**) or top-level design file (Figure 3–7) and click **Open**. The **Convert MAX+PLUS II Project** command generates a Quartus II Project File (**.qpf**) and a Quartus II Settings File (**.qsf**). The Quartus II software stores project and design assignments in the Quartus II Settings File, which is equivalent to the Assignments and Configuration File in the MAX+PLUS II software.

You also can open and convert a MAX+PLUS II design with the **Open Project** command. In the **Open Project** dialog box, browse to the Assignments and Configuration File or the top-level design file. Click **Open** to display the **Convert MAX+PLUS II Project** dialog box.

The Quartus II software can import all MAX+PLUS II-generated files, but it cannot save files in the MAX+PLUS II format. You cannot open a Quartus II project in the MAX+PLUS II software, nor can you convert a Quartus II project to a MAX+PLUS II project.

Figure 3–7. Convert MAX+PLUS II Project Dialog Box



The conversion process performs the following actions:

- Converts the MAX+PLUS II Assignments and Configuration File into a Quartus II Settings File (equivalent to importing all MAX+PLUS II assignments)
- Creates a Quartus II Project File
- Displays all errors and warnings in the Quartus II message window

The Quartus II software can read MAX+PLUS II generated Graphic Design Files (**.gdf**) and Simulation Channel Files (**.scf**) without converting them. These files are not modified during a MAX+PLUS II design conversion.

Converting MAX+PLUS II Graphic Design Files

The Quartus II Block Editor (similar to the MAX+PLUS II Graphic Editor) saves files as Block Design Files (**.bdf**). You can convert your MAX+PLUS II Graphic Design File into a Quartus II Block Design File using one of the following methods:

1. Open the Graphic Design File and on the File menu, click **Save As**. The **Save As** dialog box is shown.
2. In the **Save as type** list, select **Block Diagram/Schematic File** (*.**bdf**).

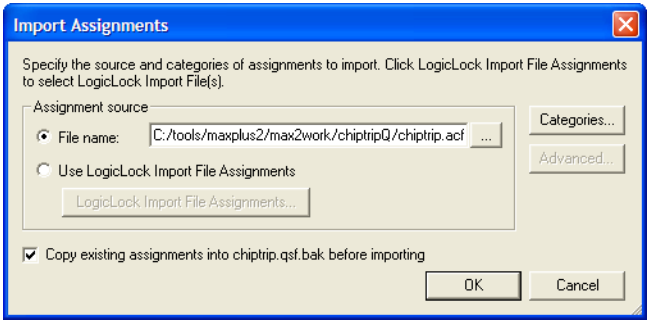
3. Run the **quartus_g2b.exe** command line executable located in the \<*Quartus II installation*>**bin** directory. For example, to convert the **chiptrip.gdf** file to a Block Design File, type the following command at a command prompt:

```
quartus_g2b.exe chip_trip.gdf ↵
```

Importing MAX+PLUS II Assignments

You can import MAX+PLUS II assignments into an existing Quartus II project. Open the project, and on the Assignments menu, click **Import Assignments**. Browse to the Assignments and Configuration File (Figure 3–8). You can also import Quartus II Settings Files and Entity Setting Files (**.esf**).

Figure 3–8. Import Assignments Dialog Box



The Quartus II software accepts most MAX+PLUS II assignments. However, some assignments can be imported incorrectly from the MAX+ PLUS II software into the Quartus II software due to differences in node naming conventions and the advanced Quartus II integrated synthesis algorithms.

The differing node naming conventions in the Quartus II and MAX+PLUS II software can cause improper mapping when importing your design from MAX+PLUS II software into the Quartus II software. Improper node names can interfere with the design logic if you are unaware of these node name differences and do not take appropriate

steps to prevent improper node name mapping. Table 3–2 compares the differences between the naming conventions used by the Quartus II and MAX+PLUS II software.

Table 3–2. Quartus II & MAX+PLUS II Node & Pin Naming Conventions

Feature	Quartus II Format	MAX+PLUS II Format
Node name	`auto_max:auto|q0`	`|auto_max:auto|q0`
Pin name	`d[0], d[1], d[2]`	`d0, d1, d2`

When you import MAX+PLUS II assignments containing node names that use numbers, such as `signal0` or `signal1`, the Quartus II software imports the original assignment and also creates an additional copy of the assignment. The additional assignment has square brackets inserted around the number, resulting in `signal[0]` or `signal[1]`. The square bracket format is legal for signals that are part of a bus, but creates illegal signal names for signals that are not part of a bus in the Quartus II software. If your MAX+PLUS II design contains node names that end in a number and are not part of a bus, you can edit the Quartus II Settings File to remove the square brackets from the node names after importing them.

You can remove obsolete assignments in the **Remove Assignments** dialog box. Open this dialog box on the Assignments menu by clicking **Remove Assignments**.

The Quartus II software may not recognize valid MAX+PLUS II node names, or may split MAX+PLUS II nodes into two different nodes. As a result, any assignments made to synthesized nodes are not recognized during compilation.

For more information about Quartus II node naming conventions, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Quartus II Design Flow

The following sections include information to help you get started using the Quartus II software. They describe the similarities and differences between the Quartus II software and the MAX+PLUS II software. The following sections highlight improvements and benefits in the Quartus II software.

Creating a New Project

The Quartus II software provides a wizard to help you create new projects. On the File menu, click **New Project Wizard** to start the New Project Wizard. The New Project Wizard generates the Quartus II Project File and Quartus II Settings File for your project.

Design Entry

The Quartus II software supports the following design entry methods:

- Altera HDL (AHDL) Text Design File (**.tdf**)
- Block Diagram File
- EDIF Netlist File (**.edf**)
- Verilog Quartus Mapping Netlist File (**.vqm**)
- VHDL (**.vhd**)
- Verilog HDL (**.v**)

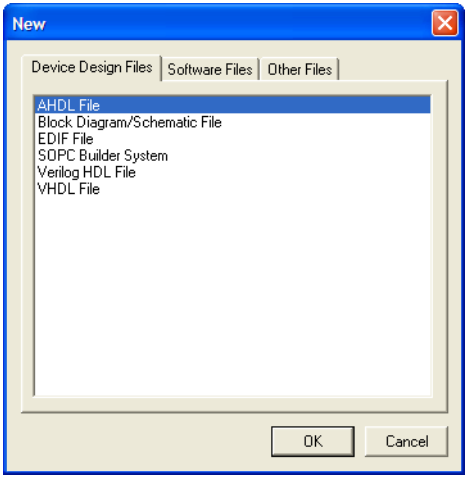
The Quartus II software has an advanced integrated synthesis engine that fully supports the Verilog HDL and VHDL languages and provides options to control the synthesis process.

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To create a new design file, perform the following steps:

1. On the File menu, click **New**. The **New** dialog box appears.
2. Click the **Device Design Files** tab.
3. Select a design entry type.
4. Click **OK** (see Figure 3–9).

Figure 3–9. New Dialog Box

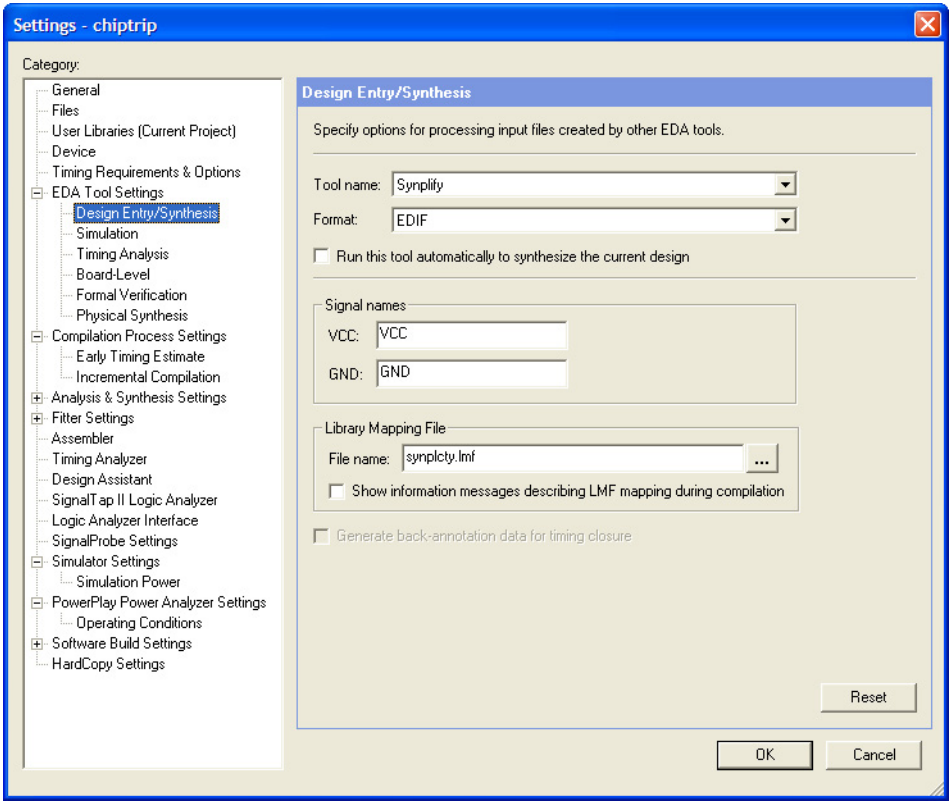


You can create other files from the **Software Files** tab and **Other Files** tab of the **New** dialog box on the File menu. For example, the Vector Waveform File (**.vwf**) is located in the **Other Files** tab.

To analyze a netlist file created by an EDA tool, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Design Entry & Synthesis**. The **Design Entry & Synthesis** page appears.
3. In the **Tool** name list, select the synthesis tool used to generate the netlist (Figure 3–10).

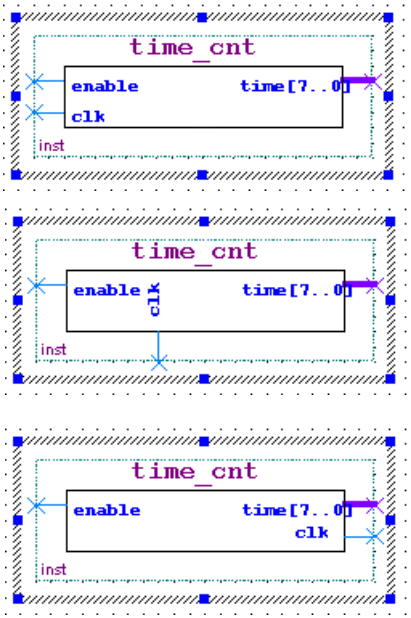
Figure 3–10. Settings Dialog Box Specifying Design Entry Tool



The Quartus II Block Editor has many advantages over the MAX+PLUS II Graphic Editor. The Block Editor offers an unlimited sheet size, multiple region selections, an enhanced Symbol Editor, and conduits.

The Symbol Editor allows you to change the positions of the ports in a symbol (refer to the three images in Figure 3–11). You can reduce wire congestion around a symbol by changing the positions of the ports.

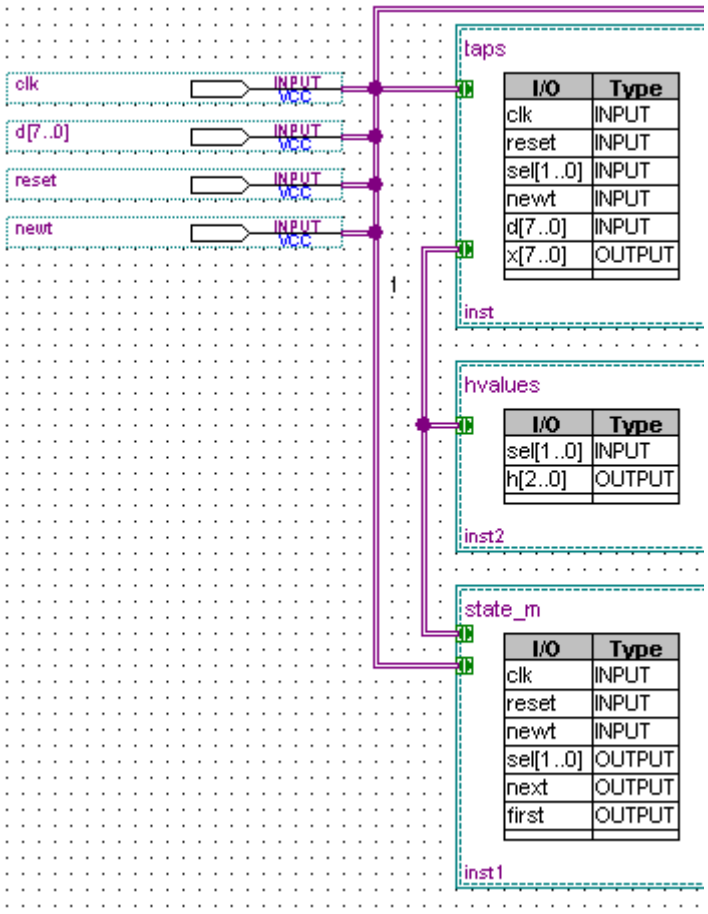
Figure 3–11. Various Port Position for a Symbol



To make changes to a symbol in a Block Design File, right-click a symbol in the Block Editor and select **Properties** to display the **Symbol Properties** dialog box. This dialog box allows you to change the instance name, add parameters, and specify the line and text color.

You can use conduits to connect blocks (including pins) in the Block Editor. Conduits contain signals for the connected objects (see Figure 3–12). You can determine the connections between various blocks in the **Conduit Properties** dialog box by right-clicking a conduit and clicking **Properties**.

Figure 3–12. Blocks & Pins Connected with Conduits



Making Assignments

The Quartus II software stores all project and design assignments in a Quartus II Settings File, which is a collection of assignments stored as Tcl commands and organized by the compilation stage and assignment type. The Quartus II Settings File stores all assignments, regardless of how they are made, from the Floorplan Editor, the Pin Planner, the Assignment Editor, with Tcl, or any other method.

Assignment Editor

The Assignment Editor is an intuitive spreadsheet interface designed to allow you to make, change, and manage a large number of assignments easily. With the Assignment Editor, you can list all available pin numbers and design pin names for efficiently creating pin assignments. You also can filter all assignments based on assignment categories and node names for viewing and creating assignments.

The Assignment Editor is composed of the Category Bar, Node Filter Bar, Information Bar, Edit Bar, and spreadsheet.

To make an assignment, follow these steps:

1. On the Assignments menu, click **Assignment Editor**. The **Assignment Editor** window appears.
2. Select an assignment category in the **Category** bar.
3. Select a node name using the Node Finder or type a node name filter into the **Node Filter** bar. (This step is optional; it excludes all assignments unrelated to the node name.)
4. Type the required values into the spreadsheet.
5. On the File menu, click **Save**.

If you are unsure about the purpose of a cell in the spreadsheet, select the cell and read the description displayed in the **Information** bar.

You can use the **Edit** bar to change the contents of multiple selected cells simultaneously. Select cells in the spreadsheet and type the value in the **Edit** box.

Other advantages of the Assignment Editor include clipboard support in the spreadsheet and automatic font coloring to identify the status of assignments.

For more information, refer to the *Assignment Editor* chapter in volume 1 of the *Quartus II Handbook.*

Timing Assignments

You can use the timing wizard to help you set your timing requirements. On the Assignments menu, click **Timing Wizard** to create global clock and timing settings. The settings include f_{MAX}, setup times, hold times, clock to output delay times, and individual absolute or derived clocks.

You also can set timing settings manually by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Setting** dialog box is shown.
2. In the **Category** list, select **Timing Requirements & Options**. The **Timing Requirements & Options** page is shown.
3. Set your timing settings.

You can make more complex timing assignments with the Quartus II software than allowed by the MAX+PLUS II software, including multicycle and point-to-point assignments using wildcards and time groups.

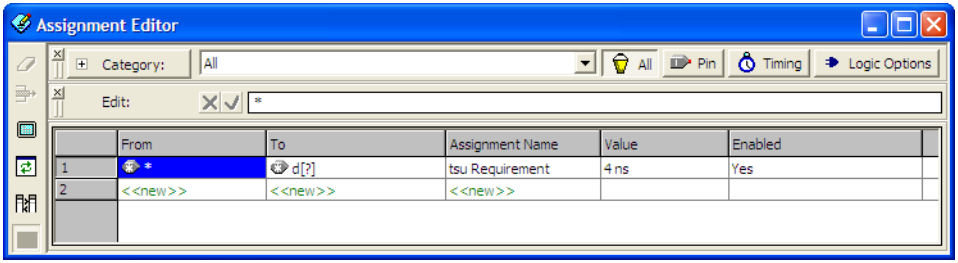
A time group is a collection of design nodes grouped together and represented as a single unit for the purpose of making timing assignments to the collection.

Multicycle timing assignments allow you to identify register-to-register paths in the design where you expect a delayed latch edge. This assignment enables accurate timing analysis of your design.

Point-to-point timing assignments allow you to specify the required delay between two pins, two registers, or a pin and a register. This assignment helps you optimize and verify your design timing requirements.

Wildcard characters "?" and " * " allow you to apply an assignment to a large number of nodes with just a few assignments. For example, Figure 3–13 shows a 4 ns t_{SU} requirement assignment to all paths from any node to the "d" bus in the Assignment Editor.

Figure 3–13. Single t_{SU} Timing Assignment Applied to All Nodes of a Bus



For more information, refer to the *Classic Timing Analyzer* or the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Synthesis

The Quartus II advanced integrated synthesis software fully supports the hardware description languages, Verilog HDL, VHDL, and AHDL, schematic entry, and also provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use, stand-alone solution for today's designs.

You can specify synthesis options in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Similar to MAX+PLUS II synthesis options, you select one of these optimization techniques: **Speed, Area,** or **Balanced**.

To achieve higher design performance, you can turn on synthesis netlist optimizations that are available when targeting certain devices. You can unmap a netlist created by an EDA tool and remap the components in the netlist back to Altera primitives by turning on **Perform WYSIWYG primitive resynthesis**. Additionally, you can move registers across combinational logic to balance timing without changing design functionality by turning on **Perform gate-level register retiming**. Both of these options are accessible from the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box on the Assignments menu.

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

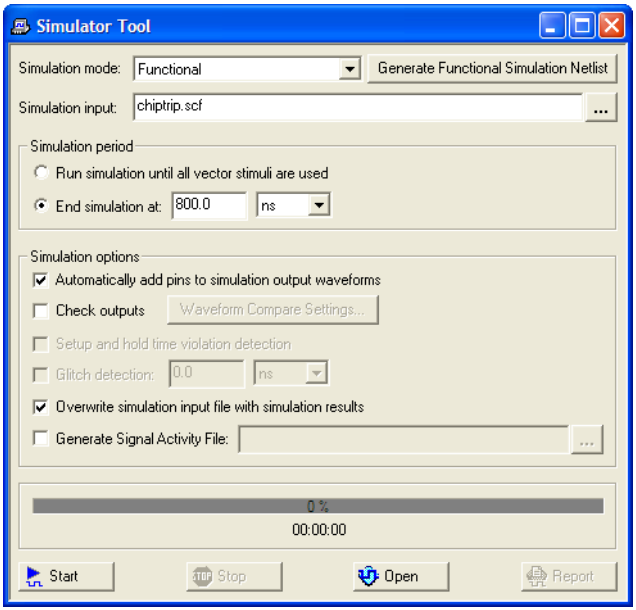
Functional Simulation

Similar to the MAX+PLUS II Simulator, the Quartus II Simulator Tool performs both functional and timing simulations.

To open the Simulator Tool, on MAX+PLUS II menu, click **Simulator** or on the Tools menu, click **Simulator Tool**. Before you perform a functional simulation, an internal functional simulation netlist is required. Click **Generate Functional Simulation Netlist** in the **Simulator Tool** window (Figure 3–14), or on the Processing menu, click **Generate Functional Simulation Netlist**.

Generating a functional simulation netlist creates a separate database that improves the performance of the simulation significantly.

Figure 3–14. Simulator Tool Dialog Box



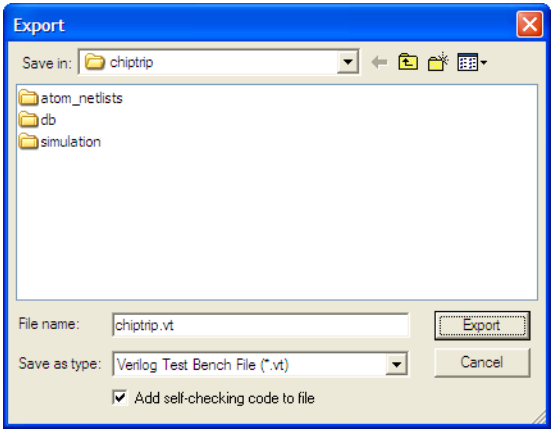
You can view and modify the simulator options on the **Simulator** page of the **Settings** dialog box or in the **Simulator Tool** window. You can set the simulation period and turn **Check outputs** on or off. You can choose to display the simulation outputs in the simulation report or in the Vector Waveform File. To display the simulation results in the simulation input vector waveform file, which is the MAX+PLUS II behavior, turn on **Overwrite simulation input file with simulation results**.

When using either the MAX+PLUS II or Quartus II software, you may have to compile additional behavioral models to perform a simulation with an EDA simulation tool. In the Quartus II software, behavioral models for library of parameterized modules (LPM) functions and Altera-specific megafunctions are available in the **altera_mf** and **220model** library files, respectively. The **220model** and **altera_mf** files can be found in the \<*Quartus II Installation*>**eda\sim_lib** directory.

The Quartus II schematic design files (Block Design File, or **.bdf**) are not compatible with EDA simulation tools. To perform a register transfer level (RTL) functional simulation of a Block Design File using an EDA tool, convert your schematic designs to a VHDL or Verilog HDL design file. Open the schematic design file and on the File menu, click **Create/Update > Create HDL Design File for Current File** to create an HDL design file that corresponds to your Block Design File.

You can export a Vector Waveform File or Simulator Channel File as a Verilog HDL or VHDL test bench file for simulation with an EDA tool. Open your Vector Waveform File or Simulator Channel File and on the File menu, click **Export**. See Figure 3–15. Select **Verilog** or **VHDL Test Bench File (*.vt)** from the **Save as type** list. Turn on **Add self-checking code to file** to add additional self-checking code to the test bench.

Figure 3–15. Export Dialog Box



Place & Route

The Quartus II PowerFit is an incremental fitter that performs place-and-route to fit your design into the targeted device. You can control the Fitter behavior with options in the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu.

High-density device families supported in the Quartus II software, such as the Stratix series, sometimes require significant fitter effort to achieve an optimal fit. The Quartus II software offers several options to reduce the time required to fit a design. You can control the effort the Quartus II Fitter expends to achieve your timing requirements with these options:

- **Optimize timing** performs timing-based placement using the timing requirements you specify for the design. You can use this option by itself or with one or more of the options below.
- **Optimize hold timing** optimizes the hold times within a device to meet timing requirements and assignments you specify. You can select this option only if the Optimize timing option is also chosen.
- **Optimize fast-corner timing** instructs the Fitter, when optimizing your design, to consider fast-corner delays, in addition to slow-corner delays, from the fast-corner timing model (fastest manufactured device, operating in low-temperature and high-voltage conditions). You can select this option only if the **Optimize timing** option is also chosen.

If minimizing compilation time is more important than achieving specific timing results, you can turn these options off.

Another way to decrease the processing time and effort the Fitter expends to fit your design is to select either **Standard Fit** or **Fast Fit** in the **Fitter Effort** box of the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu. The option you select affects the Fitter behavior and your design as described below.

- Select **Standard Fit** for the Fitter to use the highest effort and preserve the performance from previous compilations.
- Select **Fast Fit** for up to 50% faster compilation times, although this may reduce design performance.

You can also select **Auto Fit** to decrease compilation time by directing the Fitter to reduce Fitter effort after meeting your timing requirements. The **Auto Fit** option is available for select devices.

For more information, refer to the *Area & Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

To further reduce compilation times, turn on **Limit to one fitting attempt** in the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu.

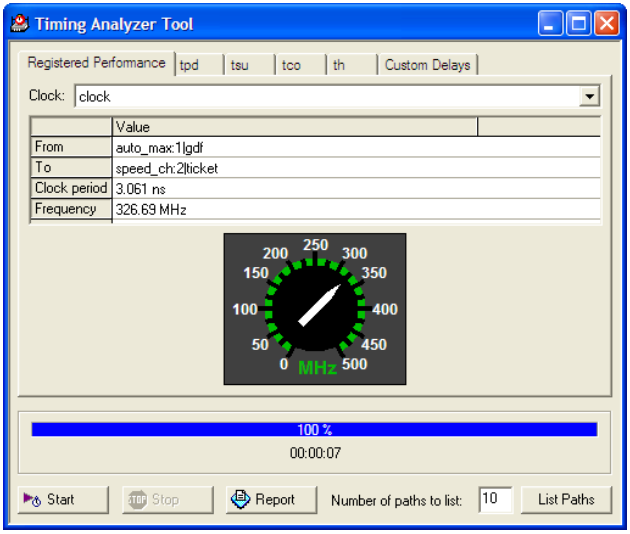
If your design is very close to meeting your timing requirements, you can control the seed number used in the fitting algorithm by changing the value in the **Seed** box of the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu. The default seed value is 1. You can specify any non-negative integer value. Changing the value of the seed only repositions the starting location of the Fitter, but does not affect compilation time or the Fitter effort level. However, if your design is difficult to fit optimally or takes a long time to fit, sometimes you can improve results or processing time by changing the seed value.

Timing Analysis

You can use the Quartus II Timing Analyzer to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer.

Launch the Timing Analyzer Tool on the MAX+PLUS II menu by clicking **Timing Analyzer** or on the Tools menu by clicking **Timing Analyzer Tool**. See Figure 3–16. To start the analysis, click **Start** in the Timing Analyzer Tool or on the Processing menu, by pointing to Start, and clicking **Start Timing Analyzer**.

Figure 3–16. Registered Performance Tab of the Timing Analyzer Tool



The Quartus II Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains. You can ignore paths that cross clock domains by using the following options in the **Timing Requirements & Options** page in the **Settings** dialog box on the Assignments menu:

- Create a **Cut Timing Path** assignment
- Turn on **Cut paths between unrelated clock domains**

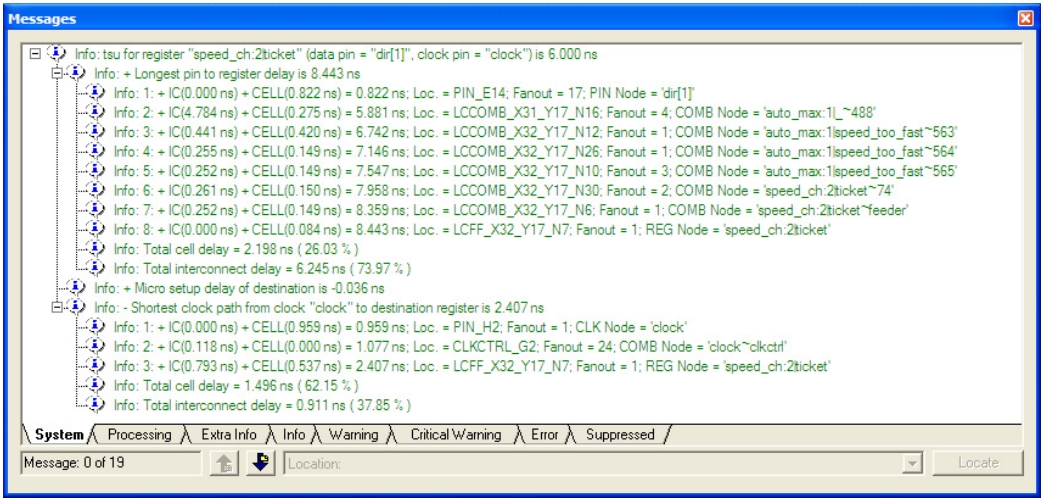
To view the results from the Timing Analyzer Tool, you can click on the **Report**, or to get specific information, click on any of the following tabs at the top of the Timing Analyzer window:

- Registered Performance
- t_{PD}
- t_{SU}
- t_{CO}
- t_{H}
- Custom Delays

The Quartus II Timing Analyzer reports both f_{MAX} and slack. Slack is the margin by which the timing requirement was met or not met. A positive slack value, displayed in black, indicates the margin by which a requirement was met. A negative slack value, displayed in red, indicates the margin by which a requirement was not met.

To analyze a particular path in more detail, select a path in the Timing Analyzer Tool and click **List Paths**. This displays a detailed description of the path in the **System** tab of the **Messages** window (Figure 3–17).

Figure 3–17. Messages Window Displaying Detailed Timing Information



For more information, refer to the *Classic Timing Analyzer* or the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Timing Closure Floorplan

The Quartus II Timing Closure Floorplan is similar to the MAX+PLUS II Floorplan Editor but has many improvements to help you more effectively view and debug your design. With its ability to display logic cell usage, routing congestion, critical paths, and LogicLock™ regions, the Timing Closure Floorplan also makes the task of improving your design performance much easier.

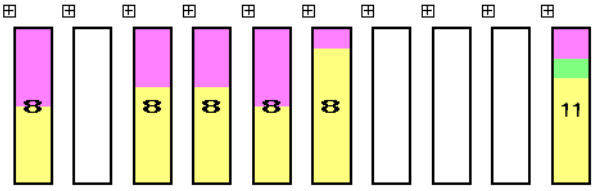
To view the Timing Closure Floorplan, on the MAX+PLUS II menu, click **Floorplan Editor** or **Timing Closure Floorplan**.

The Timing Closure Floorplan Editor provides Interior Cell views equivalent to the MAX+PLUS II logic array block (LAB) views. In addition to these views, available from the View menu, you also can select from the Interior MegaLABs (where applicable), Interior LABs, and Field views.

The Pin Planner is equivalent to the MAX+PLUS II Device view. The Pin Planner can be launched from the Timing Closure Floorplan Editor by selecting **Package** (Top or Bottom) from the View menu or on the Assignments menu by clicking **Pin Planner**.

The Interior LABs view hides cell details for logic cells, Adaptive Logic Modules (ALM), and macrocells, and shows LAB information (see Figure 3–18). You can display the number of cells used in each LAB on the View menu by clicking **Show Usage Numbers**.

Figure 3–18. Interior LAB View of the Timing Closure Floorplan



The Field view is a color-coded, high-level view of your device resources that hides both cell and LAB details. In the Field view, you can see critical paths and routing congestion in your design.

The View Critical Paths feature shows a percentage of all critical paths in your floorplan. You can enable this feature on the View menu by clicking **Show Critical Paths**. You can control the number of critical paths shown by modifying the settings in the **Critical Paths Settings** dialog box on the View menu.

The View Congestion feature displays routing congestion by coloring and shading logic resources. Darker shading shows greater resource utilization. This feature assists in identifying locations where there is a lack of routing resources.

To show lower level details in any view, right-click on a resource and click **Show Details**.

For more information, refer to the *Timing Closure Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

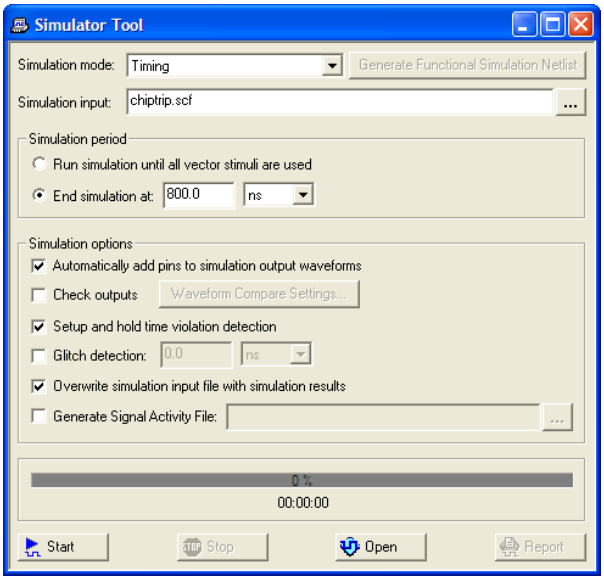
Timing Simulation

Timing simulation is an important part of the verification process. The Quartus II software supports native timing simulation and exports simulation netlists to third-party software for design verification.

Quartus II Simulator Tool

The Quartus II Simulator tool is an easy-to-use integrated solution that uses the compiler database to simulate the logical and timing performance of your design (Figure 3–19). When performing timing simulation, the simulator uses place-and-route timing information.

Figure 3–19. Quartus II Simulator Tool



You can use Vector Table Output Files (**.tbl**), Vector Waveform Files, Vector Files (**.vec**), or an existing Simulator Channel File as the vector stimuli for your simulation.

The simulation options available are similar to the options available in the MAX+PLUS II Simulator. You can control the length of the simulation and the type of checks performed by the Simulator. When the MAX+PLUS II look and feel is selected, the **Overwrite simulation input file with simulation results** option is on by default. If you turn it off, the simulation results are written to the report file. To view the report file, click **Report** in the Simulator Tool window.

EDA Timing Simulation

The Quartus II software also supports timing simulation with other EDA simulation software. Performing timing simulation with other EDA simulation software requires a Quartus II generated timing netlist file in the form of a Verilog Output File (**.vo**) or VHDL Output File (**.vho**), a Standard Delay Format Output File (**.sdo**), and a device-specific atom file (or files), shown in Table 3–3.

Table 3–3. Altera Timing Simulation Library Files

Verilog	VHDL
<*device_family*>_atoms.v	<*device_family*>_atoms_87.vhd
	<*device_family*>_atoms.vhd
	<*device_family*>_components.vhd

Specify your EDA simulation tool by performing the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. In the **Tool name list**, select your EDA Tool.

You can generate a timing netlist for the selected EDA simulator tool by running a full compile or on the Processing menu, by pointing to Start and clicking **Start EDA Netlist Writer**. The generated netlist and SDF file are placed into the \<*project directory*>**\simulation**\<*EDA simulator tool*> directory. The device-specific atom files are located in the \<*Quartus II Install*>**\eda\sim_lib** directory.

Power Estimation

To develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system, you need an accurate estimate of the power that your design consumes. You can estimate power by using the PowerPlay Early Power Estimation spreadsheet available on the Altera Web Site at **www.altera.com**, or with the PowerPlay Power Analyzer in the Quartus II software.

You can perform early power estimation with the PowerPlay Early Power Estimation spreadsheet by entering device resource and performance information. The Quartus II PowerPlay Analyzer tool performs

vector-based power analysis by reading either a Signal Activity File (**.saf**), generated from a Quartus II simulation, or a Value Change Dump File (VCD) generated from a third-party simulation.

For more information about how to use the PowerPlay Power Analyzer tool, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Programming

The Quartus II Programmer has the same functionality as the MAX+PLUS II Programmer, including programming, verifying, examining, and blank checking operations. Additionally, the Quartus II Programmer now supports the erase capability for CPLDs. To improve usability, the Quartus II Programmer displays all programming-related information in one window (Figure 3–20).

Click **Add File** or **Add Device** in the Programmer window to add a file or device, respectively.

Figure 3–20. Programmer Window

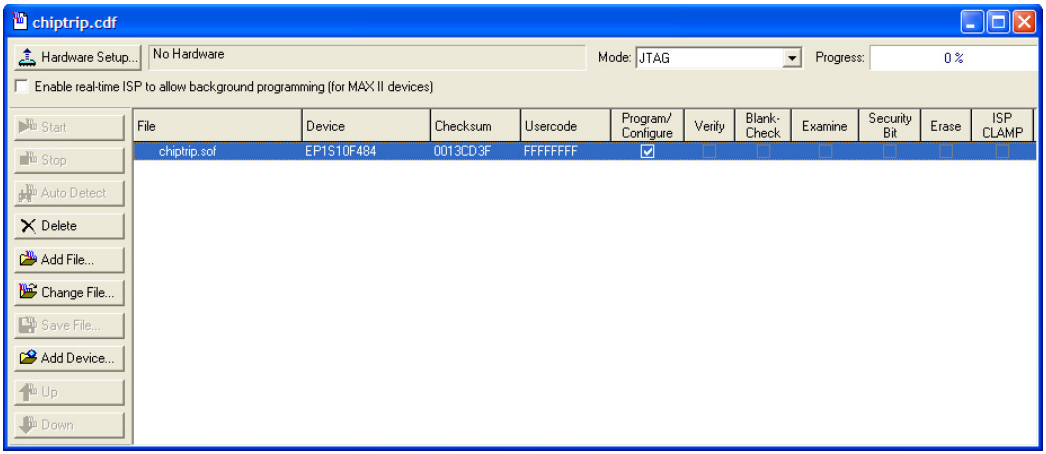


Figure 3–20 shows that the Programmer Window now supports Erase capability.

You can save the programmer settings as a Chain Description File (.**cdf**). The CDF is an ASCII text file that stores device name, device order, and programming file name information.

Conclusion

The Quartus II software is the most comprehensive design environment available for programmable logic designs. Features such as the MAX+PLUS II look and feel help you make the transition from Altera's MAX+PLUS II design software and become more productive with the Quartus II software. The Quartus II software has all the capabilities and features of the MAX+PLUS II software and many more to speed up your design cycle.

Quick Menu Reference

The commands displayed in the MAX+PLUS II Quick Menu and the Quartus II Quick Menu vary based on whichever window is active (Figures 3–21). In the following figure, the Graphic Editor window is active.

Figure 3–21. MAX+PLUS II Quick Menus in MAX+PLUS II and Quartus II Software

Quartus II Command Reference for MAX+PLUS II Users

Table 3–4 lists the commands in the MAX+PLUS II software and gives their equivalent commands in the Quartus II software.

NA means either Not Applicable or Not Available. If a command is not listed, then the command is the same in both tools.

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 1 of 10)

MAX+PLUS II Software	Quartus II Software
MAX+PLUS II Menu	
Hierarchy Display	View menu, Utility Windows, **Project Navigator**
Graphic Editor	Block Editor
Symbol Editor	Block Symbol Editor
Text Editor	Text Editor
Waveform Editor	Waveform Editor
Floorplan Editor	Assignments menu, **Timing Closure Floorplan**
Compiler	Tools menu, **Compiler Tool**
Simulator	Tools menu, **Simulator Tool**
Timing Analyzer	Tools menu, **Timing Analyzer Tool**
Programmer	Tools menu, **Programmer**
Message Processor	View menu, Utility Windows, **Messages**
File Menu	
File menu, Project, **Name** (Ctrl+J)	File menu, **Open Project** (Ctrl+J)
File menu, Project, **Set Project to Current File** (Ctrl+Shift+J)	Project menu, **Set as Top-Level Entity** (Ctrl+Shift+J) or File menu, **New Project Wizard**
File menu, Project, **Save & Check** (Ctrl+K)	Processing menu, Start, **Start Analysis & Synthesis** (Ctrl+K) *or* Processing menu, Start, **Start Analysis & Elaboration**
File menu, Project, **Save & Compile** (Ctrl+L)	Processing menu, **Start Compilation** (Ctrl+L)

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 2 of 10)

MAX+PLUS II Software	Quartus II Software
File menu, Project, **Save & Simulate** (Ctrl+Shift+L)	Processing menu, **Start Simulation** (Ctrl+I)
File menu, Project, **Compile & Simulate** (Ctrl+Shift+K)	Processing menu, **Start Compilation & Simulation** (Ctrl+Shift+K)
File menu, Project, **Archive**	Project menu, **Archive Project**
File menu, Project, <*Recent Projects*>	File menu, <*Recent Projects*>
File menu, **Delete File**	NA
File menu, **Retrieve**	NA
File menu, **Info** (Ctrl+I)	File menu, **File Properties**
File menu, **Create Default Symbol**	File menu, Create/Update, **Create Symbol Files for Current File**
File menu, **Edit Symbol**	(Block Editor) Edit menu, **Edit Selected Symbol**
File menu, **Create Default Include File**	File menu, Create/Update, **Create AHDL Include Files for Current File**
File menu, **Hierarchy Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, Hierarchy, **Up** (Ctrl+U)	Project menu, Hierarchy, **Up** (Ctrl+U)
File menu, Hierarchy, **Down** (Ctrl+D)	Project menu, Hierarchy, **Down** (Ctrl+D)
File menu, Hierarchy, **Top**	NA
File menu, Hierarchy, **Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, **MegaWizard Plug-In Manager**	Tools menu, **MegaWizard Plug-In Manager**
(Graphic Editor) File menu, **Size**	NA
(Waveform Editor) File menu, **End Time**	(Waveform Editor) Edit menu, **End Time**
(Waveform Editor) File menu, **Compare**	(Waveform Editor) View menu, **Compare to Waveforms in File**
(Waveform Editor) File menu**, Import Vector File**	File menu, **Open** (Ctrl+O)
(Waveform Editor) File menu**, Create Table File**	File menu, **Save As**
(Hierarchy Display) File menu, **Select Hierarchy**	NA
(Hierarchy Display) File menu, **Open Editor**	(Project Navigator) Double-click
(Hierarchy Display) File menu, **Close Editor**	NA
(Hierarchy Display) File menu, **Change File Type**	(Project Navigator) Select file in Files tab and select Properties on right click menu
(Hierarchy Display) File menu, **Print Selected Files**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 3 of 10)

MAX+PLUS II Software	Quartus II Software
(Programmer) File menu, **Select Programming File**	File menu, **Open**
(Programmer) File menu, **Save Programming Data As**	File menu, **Save**
(Programmer) File menu, **Inputs/Outputs**	NA
(Programmer) File menu, **Convert SRAM Object Files**	File menu, **Convert Programming Files**
(Programmer) File menu, **Archive JTAG Programming Files**	NA
(Programmer) File menu, **Create Jam or SVF File**	File menu, Create/Update, **Create JAM, SVF, or ISC File**
(Message Processor) Select Messages	NA
(Message Processor) Save Messages As	(Messages) Save Messages on right click menu
(Timing Analyzer) Save Analysis As	Processing menu, **Compilation Report** - Save Current Report on right click menu in Timing Analyzer sections
(Simulator) Create Table File	(Waveform Editor) File menu, **Save As**
(Simulator) Execute Command File	NA
(Simulator) Inputs/Outputs	NA
Edit Menu	
(Waveform Editor) Edit menu, **Overwrite**	(Waveform Editor) Edit menu, **Value**
(Waveform Editor) Edit menu, **Insert**	(Waveform Editor) Edit menu, **Insert Waveform Interval**
(Waveform Editor) Edit menu, **Align to Grid** (Ctrl+Y)	NA
(Waveform Editor) Edit menu, **Repeat**	(Waveform Editor) Edit menu, **Repeat Paste**
(Waveform Editor) Edit menu, **Grow or Shrink**	Edit menu, **Grow or Shrink** (Ctrl+Alt+G)
(Text Editor) Edit menu, **Insert Page Break**	(Text Editor) Edit menu, **Insert Page Break**
(Text Editor) Edit menu, **Increase Indent** (F2)	(Text Editor) Edit menu, **Increase Indent**
(Text Editor) Edit menu, **Decrease Indent** (F3)	(Text Editor) Edit menu, **Decrease Indent**
(Graphic Editor) Edit menu, **Toggle Connection Dot** (Double-Click)	(Block Editor) Edit menu, **Toggle Connection Dot**
(Graphic Editor) Edit menu, **Flip Horizontal**	(Block Editor) Edit menu, **Flip Horizontal**
(Graphic Editor) Edit menu, **Flip Vertical**	(Block Editor) Edit menu, **Flip Vertical**
(Graphic Editor) Edit menu, **Rotate**	(Block Editor) Edit menu, **Rotate by Degrees**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 4 of 10)

MAX+PLUS II Software	Quartus II Software
View Menu	
View menu, **Fit in Window** (Ctrl+W)	View menu, **Fit in Window** (Ctrl+W)
View menu, **Zoom In** (Ctrl+Space)	View menu, **Zoom In** (Ctrl+Space)
View menu, **Zoom Out** (Ctrl+Shift+Space)	View menu, **Zoom Out** (Ctrl+Shift+Space)
View menu, **Normal Size** (Ctrl+1)	NA
View menu, **Maximum Size** (Ctrl+2)	NA
(Hierarchy Display) View menu, **Auto Fit in Window**	NA
(Waveform Editor) View menu, **Time Range**	View menu, **Zoom**
Assign menu, **Device**	Assignments menu, **Device** *or* Assignments menu, **Settings** (Ctrl+Shift+E)
Assign menu, **Pin/Location/Chip**	Assignments menu, **Assignment Editor** - Locations category
Assign menu, **Timing Requirements**	Assignments menu, **Assignment Editor** - Timing category
Assign menu, **Clique**	Assignments menu, **Assignment Editor** - Cliques category
Assign menu, **Logic Options**	Assignments menu, **Assignment Editor** - Logic Options category
Assign menu, **Probe**	NA
Assign menu, **Connected Pins**	Assignments menu, **Assignment Editor** - Simulation category
Assign menu, **Local Routing**	Assignments menu, **Assignment Editor** - Local Routing category
Assign menu, **Global Project Device Options**	Assignments menu, **Device** - Device & Pin Options
Assign menu, **Global Project Parameters**	Assignments menu, **Settings** - Analysis & Synthesis - Default Parameters
Assign menu, **Global Project Timing Requirements**	Assignments menu, **Timing Settings**
Assign menu, **Global Project Logic Synthesis**	Assignments menu, **Settings** - Analysis & Synthesis
Assign menu, **Ignore Project Assignments**	Assignments menu, **Assignment Editor** - disable
Assign menu, **Clear Project Assignments**	Assignments menu, **Remove Assignments**
Assign menu, **Back-Annotate Project**	Assignments menu, **Back-Annotate Assignments**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 5 of 10)

MAX+PLUS II Software	Quartus II Software
Assign menu, **Convert Obsolete Assignment Format**	NA
Utilities Menu	
 Utilities menu, **Find Text** (Ctrl+F)	Edit menu, **Find** (Ctrl+F)
 Utilities menu, **Find Node in Design File** (Ctrl+B)	 Project menu, Locate, **Locate in Design File**
 Utilities menu, **Find Node in Floorplan**	 Project menu, Locate, **Locate in Timing Closure Floorplan**
Utilities menu, **Find Clique in Floorplan**	NA
Utilities menu, **Find Node Source** (Ctrl+Shift+S)	NA
Utilities menu**, Find Node Destination** (Ctrl+Shift+D)	NA
Utilities menu, **Find Next** (Ctrl+N)	 Edit menu, **Find Next** (F3)
Utilities menu, **Find Previous** (Ctrl+Shift+N)	NA
Utilities menu, **Find Last Edit**	NA
 Utilities menu, **Search and Replace** (Ctrl+R)	 Edit menu, **Replace** (Ctrl+H)
Utilities menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Utilities menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Utilities menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Utilities menu, **Analyze Timing**	NA
Utilities menu, **Clear All Timing Analysis Tags**	NA
(Text Editor) Utilities menu, **Go To** (Ctrl+G)	 Edit menu, **Go To** (Ctrl+G)
(Text Editor) Utilities menu, **Find Matching Delimiter** (Ctrl+M)	 (Text Editor) Edit, **Find Matching Delimiter** (Ctrl+M)
(Waveform Editor) Utilities menu, **Find Next Transition** (Right Arrow)	(Waveform Editor) View menu, **Next Transition** (Right Arrow)
(Waveform Editor) Utilities menu, **Find Previous Transition** (Left Arrow)	(Waveform Editor) View menu, **Next Transition** (Left Arrow)
Options Menu	
Options menu, **User Libraries**	 Assignments menu, **Settings** (Ctrl+Shift+E) Tools, Options, **Global User LIbraries**
Options menu, **Color Palette**	Tools menu, **Options**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 6 of 10)

MAX+PLUS II Software	Quartus II Software
Options menu, **License Setup**	Tools menu, **License Setup**
Options menu, **Preferences**	Tools menu, **Options**
(Hierarchy Display) Options menu, **Orientation**	NA
(Hierarchy Display) Options menu, **Compact Display**	NA
(Hierarchy Display) Options menu, **Show All Hierarchy Branches**	(Project Navigator) Expand All on right click menu
(Hierarchy Display) Options menu, **Hide All Hierarchy Branches**	NA
(Editors) Options menu, **Font**	Tools menu, **Options**
(Editors) Options menu, **Text Size**	Tools menu, **Options**
(Graphic Editor) Options menu, **Line Style**	Edit menu, **Line**
 (Graphic Editor) Options menu, **Rubberbanding**	 Tools menu, **Options**
(Graphic Editor) Options menu, **Show Parameters**	 View menu, **Show Parameter Assignments**
(Graphic Editor) Options menu, **Show Probes**	NA
(Graphic Editor) Options menu**, Show Pins/Locations/Chips**	 View menu, **Show Pin and Location Assignments**
(Graphic Editor) Options menu, **Show Clique, Timing & Local Routing Assignments**	NA
(Graphic Editor) Options menu, **Show Logic Options**	NA
 (Graphic Editor) Options menu, **Show All** (Ctrl+Shift+M)	NA
(Graphic Editor) Options menu, **Show Guidelines** (Ctrl+Shift+G)	Tools menu, **Options** - Block/Symbol Editor page
(Graphic Editor) Options menu, **Guideline Spacing**	Tools menu, **Options** - Block/Symbol Editor page
(Symbol Editors) Options menu, **Snap to Grid**	Tools menu, **Options** - Block/Symbol Editor page
(Text Editor) Options menu, **Tab Stops**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Auto-Indent**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Syntax Coloring**	NA
(Waveform Editor) Options menu, **Snap to Grid**	 View menu, **Snap to Grid**
(Waveform Editor) Options menu, **Show Grid** (Ctrl+Shift+G)	Tools menu, **Options** - Waveform Editor page
(Waveform Editor) Options menu, **Grid Size**	Edit menu, **Grid Size** - Waveform Editor page

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 7 of 10)

MAX+PLUS II Software	Quartus II Software
(Floorplan Editor) Options menu, **Routing Statistics**	NA
 (Floorplan Editor) Options menu, **Show Node Fan-In**	 View menu, Routing, **Show Fan-In**
 (Floorplan Editor) Options menu, **Show Node Fan-Out**	 View menu, Routing, **Show Fan-Out**
 (Floorplan Editor) Options menu, **Show Path**	 View menu, Routing, **Show Paths between Nodes**
(Floorplan Editor) Options menu, **Show Moved Nodes in Gray**	NA
(Simulator) Options menu, **Breakpoint**	Processing menu, Simulation Debug, **Breakpoints**
(Simulator) Options menu, **Hardware Setup**	NA
(Timing Analyzer) Options menu, **Time Restrictions**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Auto-Recalculate**	NA
(Timing Analyzer) Options menu, **Cell Width**	NA
(Timing Analyzer) Options menu, **Cut Off I/O Pin Feedback**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Clear & Reset Paths**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Read During Write Paths**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **List Only Longest Path**	NA
(Programmer) Options menu, **Sound**	NA
(Programmer) Options menu, **Programming Options**	Tools menu, **Options** - Programmer page
(Programmer) Options menu, **Select Device**	(Programmer) Edit menu, **Change Device**
(Programmer) Options menu, **Hardware Setup**	(Programmer) Edit menu, **Hardware Setup**
Symbol (Graphic Editor)	
Symbol menu, **Enter Symbol** (Double-Click)	 (Block Editor) Edit menu, **Insert Symbol** (Double-Click)
Symbol menu, **Update Symbol**	 Edit menu, **Update Symbol or Block**
Symbol menu, **Edit Ports/Parameters**	 Edit menu, **Properties**
Element (Symbol Editor)	
Element menu, **Enter Pinstub**	Double-click on edge of symbol

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 8 of 10)

MAX+PLUS II Software	Quartus II Software
Element menu, **Enter Parameters**	NA
Templates (Text Editor)	
Templates	(Text Editor) Edit menu, **Insert Template**
Node (Waveform Editor)	
Node menu, **Insert Node** (Double-Click)	Edit menu, **Insert Node or Bus** (Double-Click)
Node menu, **Enter Nodes from SNF**	Edit menu, **Insert Node** - click on Node Finder…
Node menu, **Edit Node**	Double-click on the Node
Node menu, **Enter Group**	Edit menu, **Group**
Node menu, **Ungroup**	Edit menu, **Ungroup**
Node menu, **Sort Names**	Edit menu, **Sort**
Node menu, **Enter Separator**	NA
Layout (Floorplan Editor)	
Layout menu, **Full Screen**	View menu, **Full Screen** (Ctrl+Alt+Space)
Layout menu, **Report File Equation Viewer**	View menu, **Equations**
Layout menu, **Device View** (Double-Click)	View menu, **Package Top** or View menu, **Package Bottom**
Layout menu, **LAB View** (Double-Click)	View menu, **Interior Labs**
Layout menu, **Current Assignments Floorplan**	View menu, Assignments, **Show User Assignments**
Layout menu, **Last Compilation Floorplan**	View menu, Assignments, **Show Fitter Assignments**
Processing (Compiler)	
Processing menu, **Design Doctor**	Processing menu, Start, **Start Design Assistant**
Processing menu, **Design Doctor Settings**	Assignments menu, **Settings** - Design Assistant
Processing menu, **Functional SNF Extractor**	Processing menu, **Generate Functional Simulation Netlist**
Processing menu, **Timing SNF Extractor**	Processing menu, **Start Analysis & Synthesis**
Processing menu, **Optimize Timing SNF**	NA
Processing menu, **Linked SNF Extractor**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 9 of 10)

MAX+PLUS II Software	Quartus II Software
Processing menu, **Fitter Settings**	Assignments menu, **Settings** - Fitter Settings
Processing menu, **Report File Settings**	Assignments menu, **Settings**
Processing menu, **Generate AHDL TDO File**	NA
Processing menu, **Smart Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Total Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Preserve All Node Name Synonyms**	Assignments menu, **Settings** - Compilation Process
Interfaces (Compiler)	Assignments menu, **EDA Tool Settings**
Initialize (Simulator)	
Initialize menu, **Initialize Nodes/Groups**	NA
Initialize menu, **Initialize Memory**	NA
Initialize menu, **Save Initialization As**	NA
Initialize menu, **Restore Initialization**	NA
Initialize menu, **Reset to Initial SNF Values**	NA
Node (Timing Analyzer)	
Node menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Node menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Node menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Analysis (Timing Analyzer)	
Analysis menu, **Delay Matrix**	(Timing Analyzer Tool) Delay tab
Analysis menu, **Setup/Hold Matrix**	NA
Analysis menu, **Registered Performance**	(Timing Analyzer Tool) Registered Performance tab
JTAG (Programmer)	
JTAG menu, **Multi-Device JTAG Chain**	(Programmer) Mode: JTAG
JTAG menu, **Multi-Device JTAG Chain Setup**	(Programmer) Window
JTAG menu, **Save JCF**	File menu, **Save**
JTAG menu, **Restore JCF**	File menu, **Open**
JTAG menu, **Initiate Configuration from Configuration Device**	Tools menu, **Options** - Programmer page

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 10 of 10)	
MAX+PLUS II Software	**Quartus II Software**
FLEX (Programmer)	
FLEX menu, **Multi-Device FLEX Chain**	(Programmer) Mode: Passive Serial
FLEX menu, **Multi-Device FLEX Chain Setup**	(Programmer) Window
FLEX menu, **Save FCF**	File menu, **Save**
FLEX menu, **Restore FCF**	File menu, **Open**

Document Revision History

Table 3–5 show the revision history of this document.

Table 3–5. Document Revision History		
Date & Document Version	**Changes Made**	**Summary of Changes**
November 2006 v6.1.0	Added document revision history to chapter.	
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
December 2005 v5.1.1	Minor typographic and formatting updates.	
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	
May 2005 v5.0.0	Chapter 2 was formerly Chapter 1 in version 4.2.	
Dec. 2004 v2.1	Updated for Quartus II software version 4.2. • Chapter 1 was formerly Chapter 2. • General formatting, editing updates, and figure updates. • FLEX® 600 device support added. • Assignment Editor, Timing Assignments, and Synthesis updated. • APEX II support for balanced optimization technique removed, MAX II support added. • Minor updates to Place & Route. • Tcl commands no longer supported for the Quartus II Simulator Tool. • Excel-based power calculator replaced by PowerPlay Early Power Estimation spreadsheet. • Added support for erase capability for CPLDs.	

June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	
Feb. 2004 v1.0	Initial release.	

4. Quartus II Support of HardCopy Series Devices

QII51004-6.1.0

Introduction

This chapter includes Quartus® II Support for HardCopy® II and HardCopy Stratix® devices. This chapter is divided into the following sections:

- Quartus II Support for HardCopy II Devices
- Quartus II Support for HardCopy Stratix Devices

HardCopy II Device Support

Altera® HardCopy II devices feature 1.2-V, 90 nm process technology, and provide a structured ASIC alternative to increasingly expensive multi-million gate ASIC designs. The HardCopy II design methodology offers a fast time-to-market schedule, providing ASIC designers with a solution to long ASIC development cycles. Using the Quartus II software, you can leverage a Stratix II FPGA as a prototype and seamlessly migrate your design to a HardCopy II device for production.

This document discusses the following topics:

- HardCopy II design development flow and companion devices
- HardCopy II Device Resource Guide
- Recommended Quartus II software settings
- HardCopy II Utilities menu options and functions

For more information about HardCopy II, HardCopy Stratix, and HardCopy APEX™ devices, refer to the respective device data sheets in the *HardCopy Series Handbook*.

HardCopy II Design Benefits

Designing with HardCopy II structured ASICs offers substantial benefits over other structured ASIC offerings:

- Prototyping using a Stratix II FPGA for functional verification and system development reduces total project development time
- Seamless migration from a Stratix II FPGA prototype to a HardCopy II device reduces time to market and risk
- Unified design methodology for Stratix II FPGA design and HardCopy II design reduces the need for ASIC development software
- Low up-front development cost of HardCopy II devices reduces the financial risk to your project

Quartus II Features for HardCopy II Planning

With the Quartus II software you can design a HardCopy II device using a Stratix II device as a prototype. The Quartus II software contains the following expanded features for HardCopy II device planning:

- **HardCopy II Companion Device Assignment**—Identifies compatible HardCopy II devices for migration with the Stratix II device currently selected.

  This feature constrains the pins of your Stratix II FPGA prototype making it compatible with your HardCopy II device. It also constrains the correct resources available for the HardCopy II device making sure that your Stratix II FPGA design does not become incompatible. In addition, you are still required to compile the design targeting the HardCopy II device to ensure that the design fits, routes, and meets timing.

- **HardCopy II Utilities**—The HardCopy II Utilities functions create or overwrites HardCopy II companion revisions, change revisions to use, and compare revisions for equivalency.
- **HardCopy II Advisor**—The HardCopy II Advisor helps you follow the necessary steps to successfully submit a HardCopy II design to Altera's HardCopy Design Center.

  The HardCopy II Advisor is similar to the Resource Optimization Advisor and Timing Optimization Advisor. The HardCopy II Advisor provides guidelines you can follow during development, reporting the tasks completed as well as the tasks that remain to be completed during development.

- **HardCopy II Floorplan**—The Quartus II software can show a preliminary floorplan view of your HardCopy II design's Fitter placement results.
- **HardCopy II Design Archiving**—The Quartus II software archives the HardCopy II design project's files needed to handoff the design to the HardCopy Design Center.

  This feature is similar to the Quartus II software HardCopy Files Wizard used for HardCopy Stratix and HardCopy APEX families.

- **HardCopy II Device Preliminary Timing**—The Quartus II software performs a timing analysis of HardCopy II devices based on preliminary timing models and Fitter placements. Final timing results for HardCopy II devices are provided by the HardCopy Design Center.
- **HardCopy II Handoff Report**—The Quartus II software generates a handoff report containing information about the HardCopy II design used by the HardCopy Design Center in the design review process.
- **Formal Verification**—Cadence Encounter Conformal software can now perform formal verification between the source RTL design files and post-compile gate level netlist from a HardCopy II design.

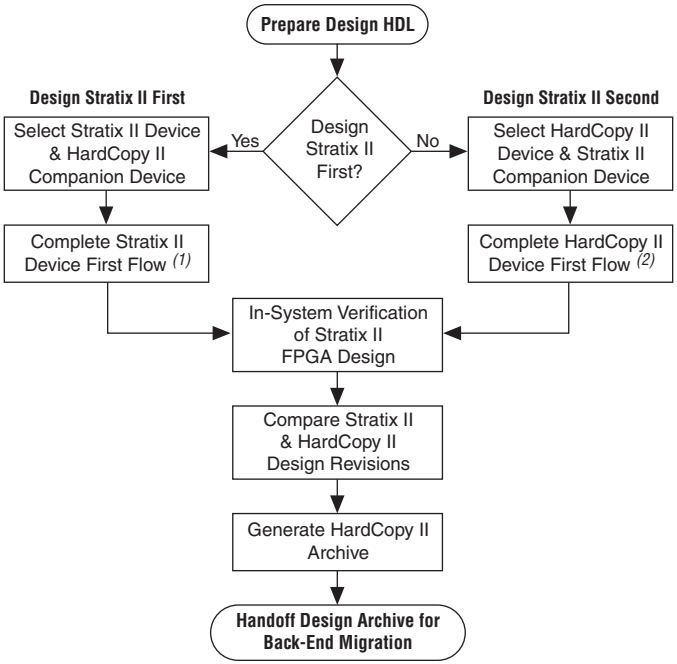
HardCopy II Development Flow

In the Quartus II software, you have two methods for designing your Stratix II FPGA and HardCopy II companion device together in one Quartus II project.

- Design the HardCopy II device first, and create the Stratix II FPGA companion device second and build your prototype for in-system verification
- Design the Stratix II FPGA first and create a HardCopy II companion device second

Both of these flows are illustrated at a high level in Figure 4–1. The added features in the HardCopy II Utilities menu assist you in completing your HardCopy II design for submission to Altera's HardCopy Design Center for back-end implementation.

Figure 4–1. HardCopy II Flow in Quartus II Software



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

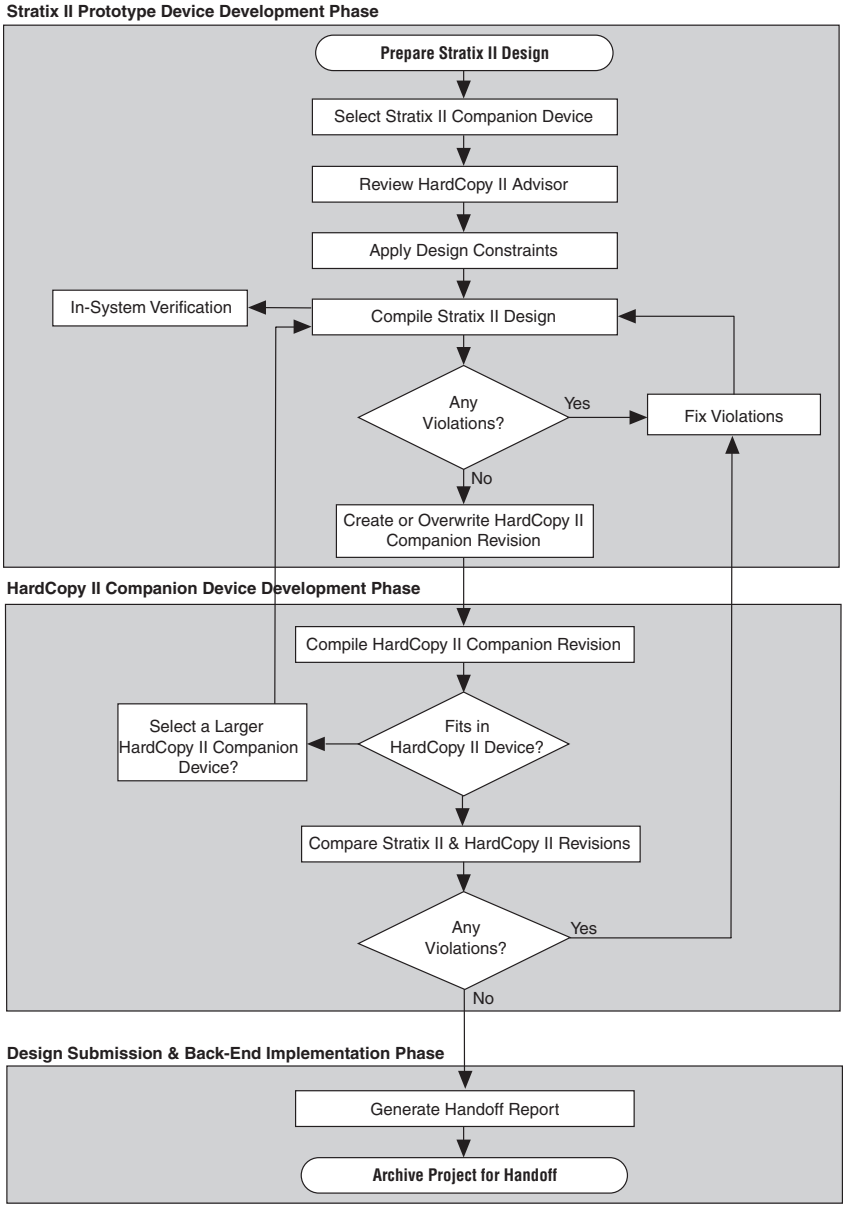
Designing the Stratix II FPGA First

The HardCopy II development flow beginning with the Stratix II FPGA prototype is very similar to a traditional Stratix II FPGA design flow, but requires a few additional tasks be performed to migrate the design to the HardCopy II companion device. To design your HardCopy II device using the Stratix II FPGA as a prototype, complete the following tasks:

- Specify a HardCopy II device for migration
- Compile the Stratix II FPGA design
- Create and compile the HardCopy II companion revision
- Compare the HardCopy II companion revision compilation to the Stratix II device compilation

Figure 4–2 provides an overview highlighting the development process for designing with a Stratix II FPGA first and creating a HardCopy II companion device second.

Figure 4–2. Designing Stratix II Device First Flow



Prototype your HardCopy II design by selecting and then compiling a Stratix II device in the Quartus II software.

After you compile the Stratix II design successfully, you can view the HardCopy II Device Resource Guide in the Quartus II software Fitter report to evaluate which HardCopy II devices meet your design's resource requirements. When you are satisfied with the compilation results and the choice of Stratix II and HardCopy II devices, on the Assignments menu, click **Settings**. In the **Category** list, select **Device**. In the **Device** page, select a HardCopy II companion device.

After you select your HardCopy II companion device, do the following:

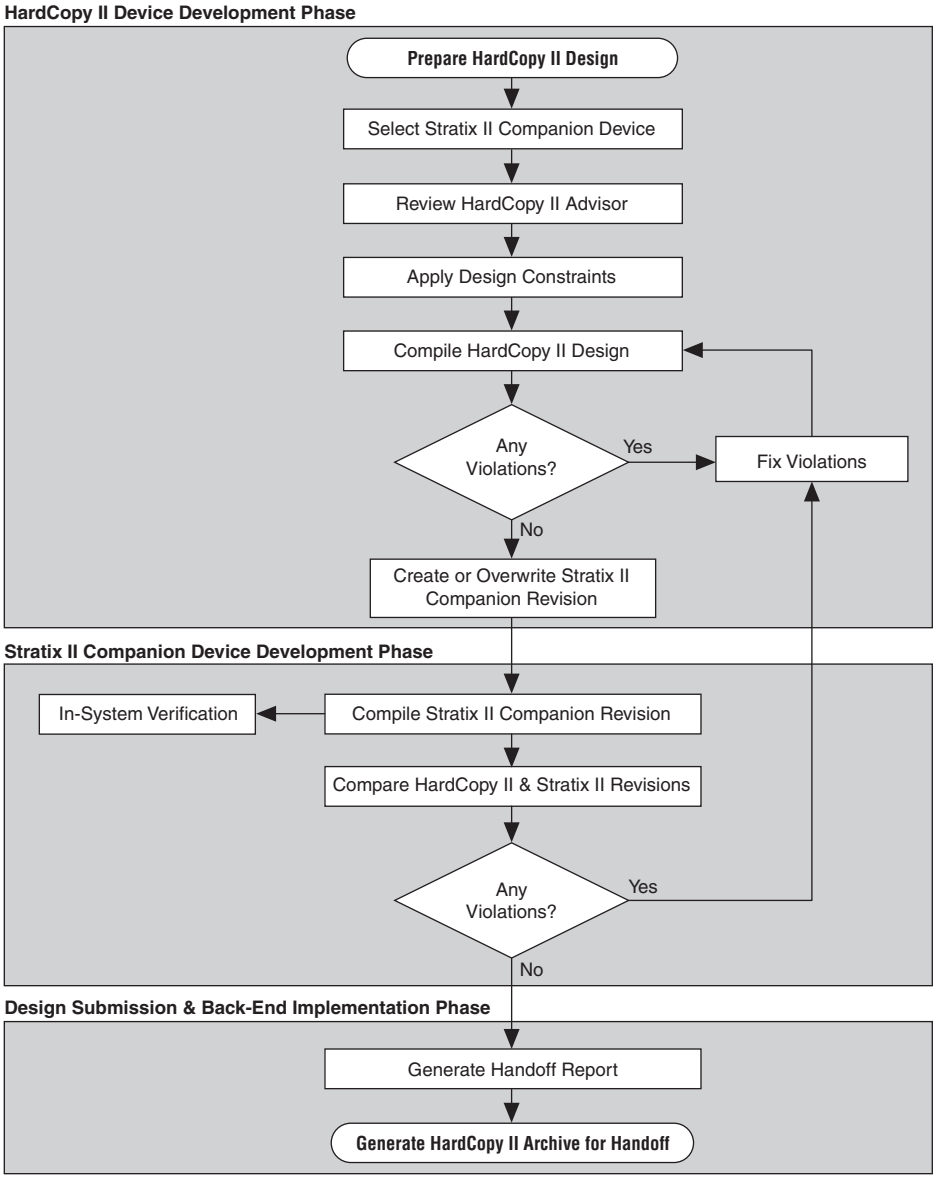
- Review the HardCopy II Advisor for required and recommended tasks to perform
- Enable Design Assistant to run during compilation
- Add timing and location assignments
- Compile your Stratix II design
- Create your HardCopy II companion revision
- Compile your design for the HardCopy II companion device
- Use the HardCopy II Utilities to compare the HardCopy II companion device compilation with the Stratix II FPGA revision
- Generate a HardCopy II Handoff Report using the HardCopy II Utilities
- Generate a HardCopy II Handoff Archive using the HardCopy II Utilities
- Arrange for submission of your HardCopy II handoff archive to Altera's HardCopy Design Center for back-end implementation

For more information about the overall design flow using the Quartus II software, refer to the *Introduction to Quartus II* manual on the Altera web site at www.altera.com.

Designing the HardCopy II Device First

The HardCopy II family presents a new option in designing unavailable in previous HardCopy families. You can design your HardCopy II device first and create your Stratix II FPGA prototype second in the Quartus II software. This allows you to see your potential maximum performance in the HardCopy II 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 traditional HardCopy II design flow where you build the FPGA first, but instead, you merely change the starting device family. The remaining tasks to complete your design for both Stratix II and HardCopy II devices roughly follow the same process (Figure 4–3). The HardCopy II Advisor adjusts its list of tasks based on which device family you start with, Stratix II or HardCopy II, to help you complete the process seamlessly.

Figure 4–3. Designing HardCopy II Device First Flow



HardCopy II Device Resource Guide

The HardCopy II Device Resource Guide compares the resources required to successfully compile a design with the resources available in the various HardCopy II devices. The report rates each HardCopy II device and each device resource for how well it fits the design. The Quartus II software generates the HardCopy II Device Resource Guide for all designs successfully compiled for Stratix II devices. This guide is found in the Fitter folder of the Compilation Report. Figure 4–4 shows an example of the HardCopy II Device Resource Guide. Refer to Table 4–1 for an explanation of the color codes in Figure 4–4.

Figure 4–4. HardCopy II Device Resource Guide

HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	Medium	None	None
2	Primary Migration Constraint		Package	Package	Package	Package	Package	Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508
4	⊟ Logic	--	19%	19%	10%	10%	6%	4%	4%
5	-- Logic cells	35572 ALUTs	--	--	--	--	--	--	--
6	-- DSP elements	0	--	--	--	--	--	--	--
7	⊟ Pins								
8	-- Total	515	515 / 302	515 / 335	515 / 493	515 / 495	515 / 699	515 / 743	515 / 952
9	-- Differential Input	0	0 / 66	0 / 70	0 / 90	0 / 90	0 / 128	0 / 224	0 / 272
10	-- Differential Output	0	0 / 44	0 / 50	0 / 70	0 / 70	0 / 112	0 / 200	0 / 256
11	-- PCI / PCI-X	0	0 / 153	0 / 167	0 / 245	0 / 247	0 / 359	0 / 367	0 / 472
12	-- DQ	0	0 / 20	0 / 20	0 / 50	0 / 50	0 / 204	0 / 204	0 / 204
13	-- DQS	0	0 / 8	0 / 8	0 / 18	0 / 18	0 / 72	0 / 72	0 / 72
14	⊟ Memory								
15	-- M-RAM	6	6 / 0	6 / 0	6 / 2	6 / 2	6 / 6	6 / 9	6 / 9
16	-- M4K blocks & M512 blocks**	44	44 / 190	44 / 190	44 / 408	44 / 408	44 / 614	44 / 816	44 / 816
17	⊟ PLLs								
18	-- Enhanced	2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 4	2 / 4	2 / 4
19	-- Fast	0	0 / 2	0 / 2	0 / 2	0 / 2	0 / 4	0 / 8	0 / 8
20	DLLs	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 2	0 / 2	0 / 2
21	⊟ SERDES								
22	-- RX	0	0 / 17	0 / 21	0 / 31	0 / 31	0 / 46	0 / 92	0 / 116
23	-- TX	0	0 / 18	0 / 19	0 / 29	0 / 29	0 / 44	0 / 88	0 / 116
24	⊟ Configuration								
25	-- CRC	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
26	-- ASMI	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
27	-- Remote Update	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
28	-- JTAG	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1

* Device is preliminary. Overall performance is expected to be degraded.
** Design contains one or more M512 blocks, which cannot be migrated to HardCopy II devices.

Use this report to determine which HardCopy II device is a potential candidate for migration of your Stratix II design. The HardCopy II device package must be compatible with the Stratix II device package. A logic

resource usage greater than 100% or a ratio greater than 1/1 in any category indicates that the design does not fit in that particular HardCopy II device.

Table 4–1. HardCopy II Device Resource Guide Color Legend

Color	Package Resource *(1)*	Device Resources
Green (High)	The design can migrate to the Hardcopy II package and the design has been fitted with target device migration enabled in the **HardCopy II Companion Device** dialog box.	The resource quantity is within the range of the HardCopy II device and the design can likely migrate if all other resources also fit. You are still required to compile the HardCopy II revision to make sure the design is able to route and migrate all other resources.
Orange (Medium)	The design can migrate to the Hardcopy II package. However, the design has not been fitted with target device migration enabled in the **HardCopy II Companion Device** dialog box.	The resource quantity is within the range of the HardCopy II device. However, the resource is at risk of exceeding the range for the HardCopy II package. If your target HardCopy II device falls in this category, compile your design targeting the HardCopy II device as soon as possible to check if the design fits and is able to route and migrate all other resources. You may need to migrate to a larger device.
Red (None)	The design cannot migrate to the Hardcopy II package.	The resource quantity exceeds the range of the HardCopy II device. The design cannot migrate to this HardCopy II device.

Note to Table 4–1:

(1) The package resource is constrained by the Stratix II FPGA for which the design was compiled. Only vertical migration devices within the same package are able to migrate to HardCopy II devices.

The HardCopy II architecture consists of an array of fine-grained HCells, which are used to build logic equivalent to Stratix II adaptive logic modules (ALMs) and digital signal processing (DSP) blocks. The DSP blocks in HardCopy II devices match the functionality of the Stratix II DSP blocks, though timing of these blocks is different than the FPGA DSP Blocks because they are constructed of HCell Macros. The M4K and M-RAM memory blocks in HardCopy II devices are equivalent to the Stratix II memory blocks. Preliminary timing reports of the HardCopy II device are available in the Quartus II software. Final timing results of the HardCopy II device are provided by the HardCopy Design Center after back-end migration is complete.

For more information about the HardCopy II device resources, refer to the *Introduction to HardCopy II Devices* and the *Description, Architecture & Features* chapters in the *HardCopy II Device Family Data Sheet* in the *HardCopy Series Handbook*.

The report example in Figure 4–4 shows the resource comparisons for a design compiled for a Stratix II EP2S130F1020 device. Based on the report, the HC230F1020 device in the 1,020-pin FineLine BGA® package is an appropriate HardCopy II device to migrate to. 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 II devices are rated red, or None, because the package types are incompatible with the Stratix II device. The 1,020-pin FBGA HC240 device is rated red because it is only compatible with the Stratix II EP2S180F1020 device.

Figure 4–5 shows the report after the (unchanged) design was recompiled with the HardCopy II HC230F1020 device specified as a migration target. Now the HC230F1020 device package and migration compatibility is rated green, or High.

Figure 4–5. HardCopy II Device Resource Guide with Target Migration Enabled

HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	High	None	None
2	Primary Migration Constraint		Package	Package	Package	Package		Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508

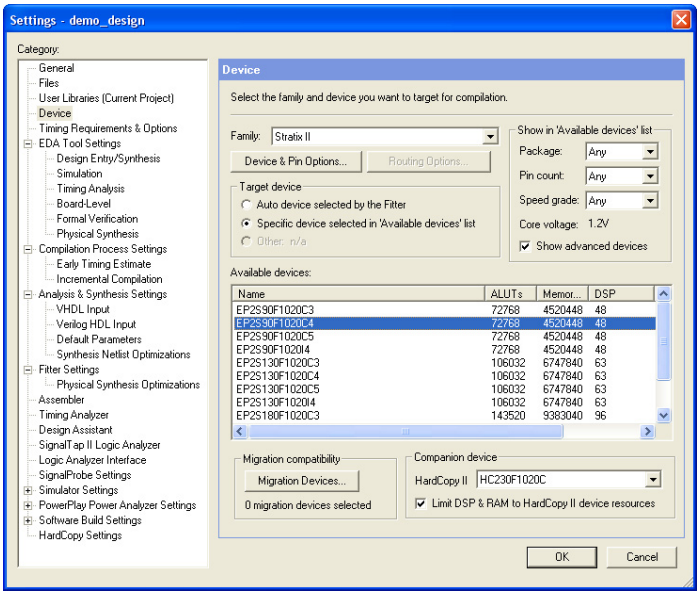
HardCopy II Companion Device Selection

In the Quartus II software, you can select a HardCopy II companion device to help structure your design for migration from a Stratix II device to a HardCopy II device. To make your HardCopy II companion device selection, on the Assignments menu, click **Settings**. In the **Settings** dialog box in the **Category** list, select **Device** (Figure 4–6) and select your companion device from the **Available devices** list.

Selecting a HardCopy II Companion device to go with your Stratix II prototype constrains the memory blocks, DSP blocks, and pin assignments, so that your Stratix II and HardCopy II devices are migration-compatible. Pin assignments are constrained in the Stratix II design revision so that the HardCopy II device selected is pin-compatible.

The Quartus II software also constrains the Stratix II design revision so it does not use M512 memory blocks or exceed the number of M-RAM blocks in the HardCopy II companion device.

Figure 4–6. Quartus II Settings Dialog Box



You can also specify your HardCopy II companion device using the following Tcl command:

```
set_global_assignment -name
DEVICE_TECHNOLOGY_MIGRATION_LIST <HardCopy II Device Part Number>
```

For example, to select the HC230F1020 device as your HardCopy II companion device for the EP2S130F1020C4 Stratix II FPGA, the Tcl command is:

```
set_global_assignment -name
DEVICE_TECHNOLOGY_MIGRATION_LIST HC230F1020C
```

HardCopy II Recommended Settings in the Quartus II Software

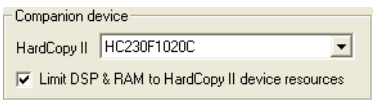
The HardCopy II development flow involves additional planning and preparation in the Quartus II software compared to a standard FPGA design. This is because you are developing your design to be implemented in two devices: a prototype of your design in a Stratix II prototype FPGA, and a companion revision in a HardCopy II device for production. You need additional settings and constraints to make the Stratix II design compatible with the HardCopy II 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 Stratix II FPGA and HardCopy II structured ASIC devices.

Limit DSP & RAM to HardCopy II Device Resources

On the Assignments menu, click **Settings** to view the **Settings** dialog box. In the **Category** list, select **Device**. In the **Family** list, select **Stratix II**. Under **Companion device**, **Limit DSP & RAM to HardCopy II device resources** is turned on by default (Figure 4–7). This maintains compatibility between the Stratix II and HardCopy II devices by ensuring your design does not use resources in the Stratix II device that are not available in the selected HardCopy II device.

If you require additional memory blocks or DSP blocks for debugging purposes using SignalTap® II, you can temporarily turn this setting off to compile and verify your design in your test environment. However, your final Stratix II and HardCopy II designs submitted to Altera for back-end migration must be compiled with this setting turned on.

Figure 4–7. Limit DSP & RAM to HardCopy II Device Resources Check Box



Enable Design Assistant to Run During Compile

You must use the Quartus II Design Assistant to check all HardCopy series 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.

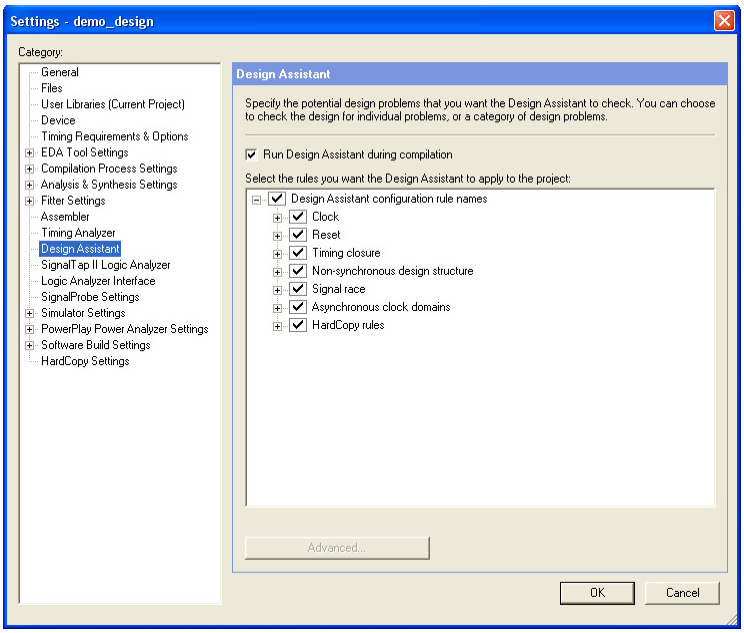
Altera recommends turning on the Design Assistant to run automatically during each compile, so that during development, you can see the violations you must fix.

For more information about the Design Assistant and the rules it uses, refer to the *Design Guidelines for HardCopy Series Devices* chapter of the *HardCopy Series Handbook*.

To enable the Design Assistant to run during compilation, on the Assignment menu, click **Settings**. In the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation** (Figure 4–8) or by entering the following Tcl command in the Tcl Console:

```
set_global_assignment -name ENABLE_DRC_SETTINGS ON
```

Figure 4–8. Enabling Design Assistant



Timing Settings

The Quartus II software version 6.1 supports two native timing analysis tools: TimeQuest Timing Analyzer and the Classic Timing Analyzer. When you specify the TimeQuest analyzer as the default timing analysis tool, the TimeQuest analyzer guides the Fitter and analyzes timing results after compilation.

TimeQuest

In Quartus II version 6.1, HardCopy Design Center strongly recommends that you use TimeQuest for timing analysis.

You can launch the TimeQuest analyzer in one of the following modes:

- Directly from the Quartus II software
- Stand-alone mode
- Command-line mode

If you want use TimeQuest for timing analysis, from the Assignments tab, in the Quartus II software, click on Timing Analysis Settings, in the new window that pops up, and click the **Use TimeQuest Timing Analyzer** during compilation tab.

Use the following Tcl command to use TimeQuest as your timing analysis engine:

```
set_instance_assignment  -name\
USE_TIMEQUEST_TIMING_ANALYZER ON
```

In order to perform a thorough Static Timing Analysis, you would need to specify all the timing requirements. The most important ones are clocks and generated clocks, input and output delays, false paths and multi-cycle paths, minimum and maximum delays.

In TimeQuest, clock latency and recovery and removal analysis are enabled by default.

For more information about TimeQuest, refer to the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* on the Altera web site at www.altera.com.

Classic Timing Analyzer

If you want use the Classic Timing Analyzer for timing analysis, from the Assignments tab, in the Quartus II software, click on Timing Analysis Settings, in the new window that pops up, click the Use Classic Timing Analyzer tab.

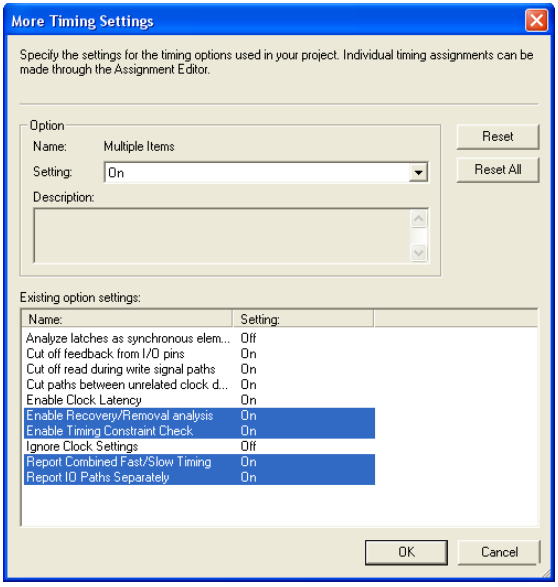
For the Classic Timing Analyzer, in the **More Timing Settings** dialog box, you can specify optional timing settings, some of which are crucial to HardCopy II development. To specify these options, on the Assignments

menu, click **Settings**. In the **Category** list, select **Timing Requirements & Options** and click **More Settings**. In the More Settings dialog box, set the desired timing settings (Figure 4–9).

For Stratix II and HardCopy II co-development, Altera recommends that you turn on the following settings when using the Classic Timing Analyzer:

- Enable Clock Latency
- Enable Recovery/Removal analysis
- Enable Timing Constraint Check
- Report Combined Fast/Slow Timing
- Report IO Paths Separately
- Enable Optimizations of the hold time along paths in the Fitter
- Enable Misc. Timing Assignments

Figure 4–9. More Timing Settings



Enable Clock Latency

Turning on the **Enable Clock Latency** option enables support for clock latency in the Timing Analyzer. Latency on a clock is a delay on the clock path and it affects clock skew. Latency is different from offset, which alters the setup relationship between two clocks.

When you enable clock latency, the design adjusts for early and late clock latency assignments. The phase-locked loop (PLL) compensation delay is analyzed as latency and does not affect the offset. For clock settings for which you have not specified an offset, the design automatically treats computed offset as latency. By using latency for these automatically calculated clock offsets, the setup relationship for registers driven by these clocks does not vary with routing. This can potentially remove the need for multicycle assignments, as well as improve results by ensuring that timing results are more consistent for each Fitter iteration.

Once enabled, you might need to add, modify, or remove these assignments for the PLL output clocks because of the potential change in the setup relationship for these clocks.

Use the following Tcl command to enable clock latency:

```
set_global_assignment -name ENABLE_CLOCK_LATENCY ON
```

Enable Recovery/Removal Analysis

This setting allows the Quartus II Timing Analysis tool to calculate recovery and removal times on control and reset signals. The recovery time is the minimum length of time that an asynchronous control input pin must be stable before the clock active edge. The removal time is the minimum length of time that an asynchronous control input pin must be stable after the clock active edge.

Altera recommends that you turn on register recovery/removal analysis in the Timing Analysis tool during development for more complete recovery/removal analysis of all logic paths in your design. However, if your design does not have a timing requirement for reset logic this option may be turned off.

Use the following Tcl command to enable recovery and removal analysis:

```
set_global_assignment -name \
ENABLE_RECOVERY_REMOVAL_ANALYSIS ON
```

Enable Timing Constraint Check

The **Enable Timing Constraint Check** setting enables the Timing Analysis tool to review your timing constraints for complete minimum and maximum timing coverage for all inputs, outputs, and bidirectional pins, as well as clock settings for all clock sources. Asynchronous pins such as resets and static control signals are also checked for minimum and maximum delay constraints. You must perform this check and review the results before handoff of the design to the HardCopy Design Center.

Use the following Tcl command to enable Timing Constraint Check:

```
set_global_assignment -name \
FLOW_ENABLE_TIMING_CONSTRAINT_CHECK ON
```

Report Combined Fast/Slow Timing

The Quartus II software can perform a separate timing analysis for worst-case and best-case conditions as independent reports. The **Report Combined Fast/Slow Timing** setting allows the Quartus II software to report slow corner delay case and fast corner delay case timing in one combined report. This setting provides a better timing report for your design by allowing you to see all hold-time issues and setup issues in one report. This report is required for HardCopy II device development. Turning on the **Report Combined Fast/Slow Timing** setting requires the Quartus II software to run the Timing Analyzer twice, once for the fast corner delay model and once for the slow corner delay model.

Use the following Tcl command to enable the **Report Combined Fast/Slow Timing** setting:

```
set_global_assignment -name DO_COMBINED_ANALYSIS ON
```

Report IO Paths Separately

Turn on the **Report IO Paths Separately** setting to create separate report panels for I/O paths constrained by the `INPUT_MAX_DELAY`, `INPUT_MIN_DELAY`, `OUTPUT_MAX_DELAY`, or `OUTPUT_MIN_DELAY` parameters. To specify these constraints, on the Assignments menu, click **Assignment Editor**. By default, I/O paths are reported in the **Clock Setup** and **Clock Hold** sections of the **Timing Analyzer** compilation report.

Altera recommends that you turn on the **Report IO Paths Separately** setting to make it easier to view the I/O timing analysis reports for each device pin. This setting is optional in FPGA designs, but is helpful for HardCopy II development because the I/O timing requirements you specify must be met in both Stratix II I/O timing and HardCopy II I/O timing results. This setting helps to guarantee drop-in compatibility between your Stratix II FPGA prototype and your HardCopy II structured ASIC.

Use the following Tcl command to enable the **Report IO Paths Separately** setting:

```
set_global_assignment -name \
REPORT_IO_PATHS_SEPARATELY ON
```

Enable Optimizations of the Hold Time Along All Paths In The Fitter

You must enable optimizations of the hold time along all paths in the fitter to allow the fitter to optimize the hold time by adding delay to the appropriate paths, including internal core transfers. You must enable the Optimize Timing option to successfully enable optimizations of the hold time.

In order to optimize hold time along all paths, select the **Fitter Settings** page of the **Settings** dialog box, click the **Optimize** hold timing arrow and select **All Paths**.

Enable Miscellaneous Timing Assignments

To achieve a smooth conversion of the Quartus II timing analysis assignments to the ones used by the HardCopy Design Center, it is recommended that you turn on the following assignments:

- Cut off feedback from I/O pins
- Cut off read during write signal paths
- Cut off clear and preset signal paths
- Default hold multicycle

You can enable these by clicking on the **More Settings** dialog box available from the Timing Requirements & Options Page of the Settings dialog box and selecting **ON** from the pull-down menu for: **Cut off feedback from I/O pins**, **Cut off read during write signal paths**, **Cut off clear and preset signal paths** and **Same as Multicycle for default hold multicycle**.

An easier way to do these would be to open the HardCopy II Advisor and then Set Up Timing Constraints, and click on **Correct All Timing Constraints** for all the above.

Quartus II Software Features Supported for HardCopy II Designs

The Quartus II software supports optimization features for HardCopy II prototype development including:

- Physical Synthesis Optimization
- LogicLock Regions
- PowerPlay Power Analyzer
- Incremental Compilation (Synthesis and Fitter)
- Maximum Fan-Out Assignments

Physical Synthesis Optimization

To enable the Physical Synthesis Optimizations for the Stratix II FPGA revision of the design, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Fitter Settings**. These optimizations are migrated into the HardCopy II companion revision for placement and timing closure. When designing with a HardCopy II device first, physical synthesis optimizations can be enabled for the HardCopy II device, and these post-fit optimizations are migrated to the Stratix II FPGA revision.

LogicLock Regions

The use of LogicLock Regions in the Stratix II FPGA is supported for designs migrating to HardCopy II. However, the LogicLock Regions are not passed into the HardCopy II Companion Revision. You can use LogicLock in the HardCopy II design but you must create new LogicLock Regions in the HardCopy II companion revision. In addition, LogicLock Regions in HardCopy II devices can not have their properties set to Auto Size. However, Floating LogicLock regions are supported. HardCopy II LogicLock Regions must be manually sized and placed in the floorplan. When LogicLock Regions are created in a HardCopy II device, they start with width and height dimensions set to (1,1), and the origin coordinates for placement are at X1_Y1 in the lower left corner of the floorplan. You must adjust the size and location of the LogicLock Regions you created in the HardCopy II device before compiling the design.

For information about using LogicLock Regions, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook* on the Altera web site at www.altera.com.

PowerPlay Power Analyzer

You can perform power estimation and analysis of your HardCopy II and Stratix II devices using the PowerPlay Early Power Estimator. Use the PowerPlay Power Analyzer for more accurate estimation of your device's power consumption. The PowerPlay Early Power Estimatior is available in the Quartus II software version 5.1 and later. The PowerPlay Power Analyzer supports HardCopy II devices in version 6.0 and later of the Quartus II software.

For more information about using the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook* on the Altera web site at www.altera.com.

Incremental Compilation

The use of the Quartus II Incremental Compilation in the Stratix II FPGA is supported when migrating a design to a HardCopy II device. Incremental compilation is supported in the Stratix II First design flow or HardCopy II First design flow.

To take advantage of Quartus II Incremental Compilation, organize your design into logical and physical partitions for synthesis and fitting (or place and route). Incremental compilation preserves the compilation results and performance of unchanged partitions in your design. This feature dramatically reduces your design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. You can also target optimization techniques, such as physical synthesis, to specific partitions while leaving other partitions untouched.

In addition, be aware of the following guidelines:

- User partitions and synthesis results are migrated to a companion device.
- LogicLock regions are suggested for user partitions, but are not migrated automatically.
- The first compilation after migration to a companion device requires a full compilation (all partitions are compiled), but subsequent compilations can be incremental if changes to the source RTL are not required. For example, PLL phase changes can be implemented incrementally if the blocks are partitioned.
- The entire design must be migrated between Stratix II and HardCopy II companion devices. The Quartus II software does not support migration of partitions between companion devices.

- Bottom-up Quartus II Incremental Compilation is not supported for HardCopy II devices.
- Physical Synthesis can be run on individual partitions within the originating device only. The resulting optimizations are preserved in the migration to the companion device.

For information about using Quartus II Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Maximum Fanout Assignments

This feature is supported beginning in Quartus II 6.1. In order to meet timing, it may be necessary to limit the number of fanouts of a net in your design. You can limit the maximum fanout of a given net by using this feature.

For example, you can use the following Tcl command to enable the maximum fanout setting:

```
set_instance_assignment -name MAX_FANOUT<number> - to\
<net name>
```

For example, if you want to limit the maximum fanout of net called "m3122_combout_1" to 25, the Tcl command is as follows:

```
set_instance_assignment -name MAX_FANOUT 25 -to\
m3122_combout_1
```

Performing ECOs with Change Manager and Chip Planner

As designs grow larger and larger in density, the need to analyze the design for performance, routing congestion, logic placement, and executing Engineering Change Orders (ECOs) becomes critical. In addition to design analysis, you can use various bottom-up and top-down flows to implement and manage the design. This becomes difficult to manage since ECOs are often implemented as last minute changes to your design.

With the Altera® Chip Planner tool, you can shorten the design cycle time significantly. When changes are made to your design as ECOs, you do not have to perform a full compilation in the Quartus II software. Instead, you would 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 (if you are using an Stratix II FPGA as a prototype). Once

the fix has been verified on the Stratix II FPGA, switch to the HardCopy II revision, apply the same ECOs, run the timing analyzer and assembler, perform a revision compare and then run the HardCopy II Netlist Writer for design submission.

There are three scenarios from a migration point of view as follows:

1. There are changes which can map one-to-one (that is, the same change can be implemented on each architecture - Stratix II FPGA and HardCopy II).
2. There are changes that must be implemented differently on the two architectures to achieve the same result.
3. There are some changes that cannot be implemented on both architectures.

The following sections outline the methods for migrating each of these types of changes.

Migrating One-to-One Changes

One-to-one changes are implemented using identical commands in both architectures. In general, such changes include those that affect only I/O cells or PLL cells. Some examples of one-to-one changes are changes such as creating, deleting or moving pins, changing pin or PLL properties, or changing pin connectivity (provided the source and destination of the connectivity changes are I/O's or PLLs) can be implemented identically on both architectures.

If such changes are exported to Tcl, a direct reapplication of the generated Tcl script (with a minor text edit) on the companion revision should implement the appropriate changes as follows:

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

A partial list of examples of this type are as follows:

- I/O creation, deletion and moves
- I/O property changes (for example, I/O standards, delay chain settings etc.)
- PLL property changes
- Connectivity changes between non-LCELL_COMB atoms (for example, PLL to I/O, DSP to I/O etc.)

Migrating Changes that must be Implemented Differently

Some changes must be implemented differently on the two architectures. Changes affecting the logic of the design may fall into this category. Examples are LUTMASK changes, LC_COMB/HSADDER creation and deletion, and connectivity changes not covered in the previous section.

The following table summarizes suggested implementation for various changes:

Table 4–2. Implementation Suggestions for Various Changes

Change Type	Suggested Implementation
Lutmask changes	Because a single Stratix II atom may require multiple HardCopy II atoms to implement, it may be necessary to change multiple HardCopy II atoms to implement the change, including adding or modifying connectivity
Make/Delete LC_COMB	If you are using a Stratix II LC_COMB in extended mode (7-LUT) or using a SHARE chain, you must create multiple atoms to implement the same logic functions in HardCopy II. Additionally, the placement of the LC_COMB cell has no meaning in the companion revision as the underlying resources are different.
Make/Delete LC_FF	The basic creation and deletion is the same on both architectures. However, as with LC_COMB creation and deletion, the location of an LC_FF in a HardCopy II revision has no meaning in the Stratix II revision, and vice versa.
Editing Logic Connectivity	Because a Stratix II LCELL_COMB atom may have to be broken up into several HardCopy II LCELL_COMB atoms, the source or destination ports for connectivity changes may need to be reanalyzed to properly implement the change in the companion revision.

Changes that Cannot be Migrated

A small set of changes cannot be implemented in the other architecture because they do not make sense in the other architecture. The best example of this occurs when moving logic in a design; because the logic fabric is different between the two architectures, locations in Stratix II make no sense in HardCopy II, and vice versa.

Overall Migration Flow

This section outlines the migration flow and the suggested procedure for implementing changes in both revisions to ensure a successful Revision Compare such that the design can be submitted to the HardCopy Design Center.

Preparing the Revisions

The general procedure for migrating changes between devices is the same, whether going from Stratix II to HardCopy II, or vice versa. The major steps are as follows:

1. Compile the design on the initial device.
2. Migrate the design from the initial device to the target device in the companion revision.
3. Compile the companion revision.
4. Perform a Revision Compare operation. The two revisions should pass the Revision Compare.

If testing identifies problems requiring ECO changes, equivalent changes can be applied to both Stratix II and HardCopy II revisions, as described in the next section.

Applying ECO Changes

The general flow for applying equivalent changes in companion revisions is as follows:

1. Make changes in one revision using the Chip Planner tools (Chip Planner, Resource Property Editor, and Change Manager), then verify and export these changes. The procedure for doing this is as follows:
   a. Make changes using the Chip Planner tool.
   b. Perform a netlist check using the Check and Save All Netlist Changes command.
   c. Verify correctness using timing analysis, simulation, and prototyping (Stratix II only). If more changes are required, repeat steps a-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. Perform a Revision Compare operation. The revisions should now match once again.

5. Verify the correctness of all changes (You may need to run timing analysis).

6. Run the HardCopy II Assembler and the HardCopy II Netlist Writer for design submission along with handoff files.

The Tcl command for running the HardCopy II Assembler is as follows:

```
execute_module -tool asm -args "--read_settings_files=\
off --write_settings_files=off"
```

The Tcl command for the HardCopy II Netlist Writer is as follows:

```
execute_module -tool cdb  -args "--generate_hardcopyii_files"\
```

For more information about using Chip Planner, refer to the *Design Analysis & Engineering Change Management with Chip Planner* chapter, in Volume 3 of the Quartus II Handbook on the Altera web site at www.altera.com.

Formal Verification of Stratix II & HardCopy II Revisions

Third party formal verification software is available for your HardCopy II design. Cadence Encounter Conformal verification software is used for Stratix II and HardCopy II families, as well as several other Altera product families.

In order to use the Conformal software with the Quartus II software project for your Stratix II and HardCopy II design revisions, you must enable the **EDA Netlist Writer.** It is necessary to turn on the EDA Netlist Writer so it can generate the necessary netlists and command files needed to run the Conformal software. To automatically run the EDA Netlist Writer during the compile of your Stratix II and HardCopy II design revisions, perform the following steps:

1. On the Assignment menu, click **EDA Tool Settings**. The **Settings** dialog box displays.

2. In the **EDA Tool Settings** list, select **Formal Verification**, and in the **Tool name** list, select **Conformal LEC**.

3. Compile your Stratix II and Hardcopy II design revisions, with both the EDA Tool Settings and the Conformal LEC turned on so the EDA Netlist Writer automatically runs.

The Quartus II EDA Netlist Writer produces one netlist for Stratix II when it is run on that revision, and generates a second netlist when it runs on the HardCopy II revision. You can compare your Stratix II post-compile netlist to your RTL source code using the scripts generated by the EDA Netlist Writer. Similarly, you can compare your HardCopy II post-compile netlist to your RTL source code with scripts provided by the EDA Netlist Writer.

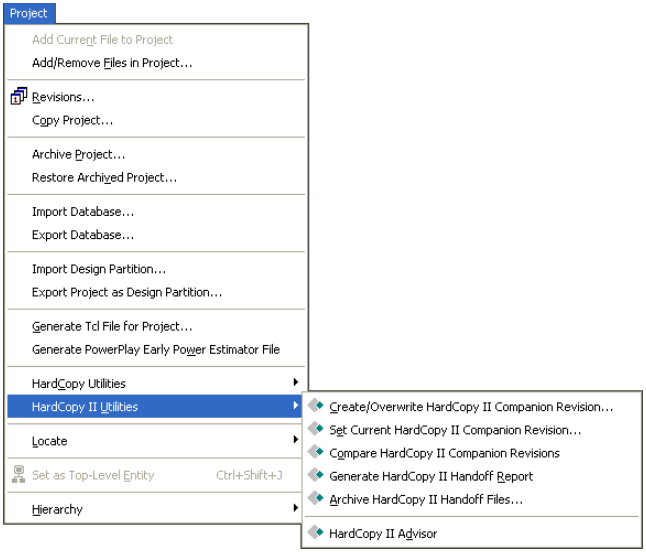
For more information about using the Cadence Encounter Conformal verification software, refer to the *Cadence Encounter Conformal Support* chapter in volume 3 of the *Quartus II Handbook*.

HardCopy II Utilities Menu

The **HardCopy II Utilities** menu is shown in the Quartus II software (Figure 4–10). To access this menu, on the Project menu, click **HardCopy II Utilities**. This menu contains the main functions you use to develop your HardCopy II design and Stratix II FPGA prototype companion revision. From the HardCopy II Utilities menu, you can:

- Create or update HardCopy II companion revisions
- Set which HardCopy II companion revision is the current revision
- Generate HardCopy II Handoff Report for design reviews
- Archive HardCopy II Handoff Files for submission to the HardCopy Design Center
- Compare the companion revisions for functional equivalence
- Track your design progress using the HardCopy II Advisor

Figure 4–10. HardCopy II Utilities Menu



Each of the features within the **HardCopy II Utilities** is summarized in Table 4–3. The process for using each of these features is explained in the following sections.

Table 4–3. HardCopy II Utilities Menu Options

Menu	Description	Applicable Design Revision	Restrictions
Create/Overwrite HardCopy II Companion Revision	Create a new companion revision or update an existing companion revision for your Stratix II and HardCopy II design.	Stratix II prototype design and HardCopy II Companion Revision	• Must disable Auto Device selection • Must set a Stratix II device and a HardCopy II companion device
Set Current HardCopy II Companion Revision	Specify which companion revision to associate with current design revision.	Stratix II prototype design and HardCopy II Companion Revision	Companion Revision must already exist
Compare HardCopy II Companion Revisions	Compares the Stratix II design revision with the HardCopy II companion design revision and generates a report.	Stratix II prototype design and HardCopy II Companion Revision	Compilation of both revisions must be complete
Generate HardCopy II Handoff Report	Generate a report containing important design information files and messages generated by the Quartus II compile	Stratix II prototype design and HardCopy II Companion Revision	• Compilation of both revisions must be complete • Compare HardCopy II Companion Revisions must have been executed
Archive HardCopy II Handoff Files	Generate a Quartus II Archive File specifically for submitting the design to the HardCopy Design Center. Similar to the HardCopy Files Wizard for HardCopy Stratix and APEX.	HardCopy II Companion Revision	• Compilation of both revisions must be completed • Compare HardCopy II Companion Revisions must have been executed • Generate HardCopy Handoff Report must have been executed
HardCopy II Advisor	Open an Advisor, similar to the Resource Optimization Advisor, helping you through the steps of creating a HardCopy II project.	Stratix II prototype design and HardCopy II Companion Revision	None

Companion Revisions

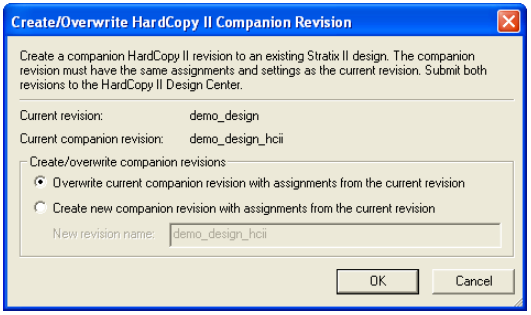
HardCopy II designs follow a different development flow in the Quartus II software compared with previous HardCopy families. You can create multiple revisions of your Stratix II prototype design, but you can also create separate revisions of your design for a HardCopy II device.

The Quartus II software creates specific HardCopy II design revisions of the project in conjunction to the regular project revisions. These parallel design revisions for HardCopy II devices are called companion revisions.

Although you can create multiple project revisions, Altera recommends that you maintain only one Stratix II FPGA revision once you have created the HardCopy II *companion revision*.

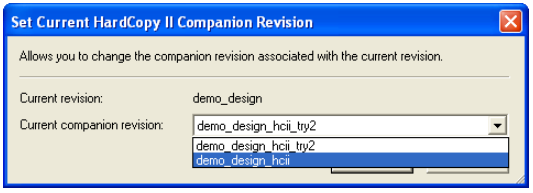
When you have successfully compiled your Stratix II prototype FPGA, you can create a HardCopy II companion revision of your design and proceed with compiling the HardCopy II companion revision. To create a companion revision, on the Project menu, point to HardCopy II Utilities and click **Create/Overwrite HardCopy II Companion Revision**. Use the dialog box to create a new companion revision or overwrite an existing companion revision (Figure 4–11).

Figure 4–11. Create or Overwrite HardCopy II Companion Revision



You can associate only one Stratix II revision to one HardCopy II companion revision. If you created more than one revision or more than one companion revision, set the current companion for the revision you are working on. On the Project menu, point to HardCopy II Utilities and click **Set Current HardCopy II Companion Revision** (Figure 4–12).

Figure 4–12. Set Current HardCopy II Companion Revision

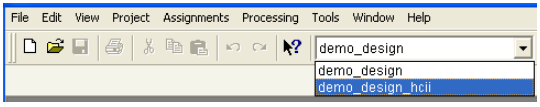


Compiling the HardCopy II Companion Revision

The Quartus II software enables you to compile your HardCopy II design with preliminary timing information. The timing constraints for the HardCopy II companion revision can be the same as the Stratix II design used to create the revision. The Quartus II software contains preliminary timing models for HardCopy II devices and you can gauge how much performance improvement you can achieve in the HardCopy II device compared to the Stratix II FPGA. Altera verifies that the HardCopy II Companion Device timing requirements are met in the HardCopy Design Center.

After you create your HardCopy II companion revision from your compiled Stratix II design, select the companion revision in the Quartus II software design revision drop-down box (Figure 4–13) or from the **Revisions** list. Compile the HardCopy II companion revision. After the Quartus II software compiles your design, you can perform a comparison check of the HardCopy II companion revision to the Stratix II prototype revision.

Figure 4–13. Changing Current Revision



Comparing HardCopy II & Stratix II Companion Revisions

Altera uses the companion revisions in a single Quartus II project to maintain the seamless migration of your design from a Stratix II FPGA to a HardCopy II structured ASIC. This methodology allows you to design with one set of Register Transfer Level (RTL) code to be used in both Stratix II FPGA and HardCopy II structured ASIC, guaranteeing functional equivalency.

When making changes to companion revisions, use the Compare HardCopy II Companion Revisions feature to ensure that your Stratix II design matches your HardCopy II design functionality and compilation settings. To compare companion revisions, on the Project menu, point to HardCopy II Utilities and click **Compare HardCopy II Companion Revisions**.

You must perform this comparison after both Stratix II and HardCopy II designs are compiled in order to hand off the design to Altera's HardCopy Design Center.

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

Generate HardCopy II Handoff Report

In order to submit a design to the HardCopy Design Center, you must generate a HardCopy II Handoff Report providing important information about the design that you want the HardCopy Design Center to review. To generate the HardCopy II Handoff Report, you must:

- Successfully compile both Stratix II and HardCopy II revisions of your design
- Successfully run the Compare HardCopy II Companion Revisions utility

Once you generate the HardCopy II Handoff Report, you can archive the design using the Archive HardCopy II Handoff Files utility described in "Archive HardCopy II Handoff Files" on page 4–33.

Archive HardCopy II Handoff Files

The last step in the HardCopy II design methodology is to archive the HardCopy II project for submission to the HardCopy Design Center for back-end migration. The HardCopy II archive utility creates a different Quartus II Archive File than the standard Quartus II project archive utility generates. This archive contains only the necessary data from the Quartus II project needed to implement the design in the HardCopy Design Center.

In order to use the **Archive HardCopy II Handoff Files** utility, you must complete the following:

- Compile both the Stratix II and HardCopy II revisions of your design
- Run the Compare HardCopy II Revisions utility
- Generate the HardCopy II Handoff Report

To select this option, on the Project menu point to HardCopy II Utilities and click **Archive HardCopy II Handoff File** utility.

HardCopy II Advisor

The HardCopy II Advisor provides the list of tasks you should follow to develop your Stratix II prototype and your HardCopy II design. To run the HardCopy II Advisor, on the Project menu, point to HardCopy II Utilities and click **HardCopy II Advisor**. The following list highlights the

checkpoints that the HardCopy II Advisor reviews. This list includes the major check points in the design process; it does not show every step in the process for completing your Stratix II and HardCopy II designs:

1. Select a Stratix II device.
2. Select a HardCopy II device.
3. Turn on the **Design Assistant**.
4. Set up timing constraints.
5. Check for incompatible assignments.
6. Compile and check Stratix II design.
7. Create or overwrite companion revision.
8. Compile and check HardCopy II companion results.
9. Compare companion revisions.
10. Generate Handoff Report.
11. Archive Handoff Files and send to Altera.

The HardCopy II Advisor shows the necessary steps that pertain to your current selected device. The Advisor shows a slightly different view for a design with Stratix II selected as compared to a design with HardCopy II selected.

In the Quartus II software, you can start designing with the HardCopy II device selected first, and build a Stratix II companion revision second. When you use this approach, the HardCopy II Advisor task list adjusts automatically to guide you from HardCopy II development through Stratix II FPGA prototyping, it then completes the comparison archiving and handoff to Altera.

When your design uses the Stratix II FPGA as your starting point, Altera recommends following the Advisor guidelines for your Stratix II FPGA until you complete the prototype revision.

When the Stratix II FPGA design is complete, create and switch to your HardCopy II companion revision and follow the Advisor steps shown in that revision until you are finished with the HardCopy II revision and are ready to submit the design to Altera for back-end migration.

Each category in the HardCopy II Advisor list has an explanation of the recommended settings and constraints, as well as quick links to the features in the Quartus II software that are needed for each section. The HardCopy II Advisor displays:

- A green check box when you have successfully completed one of the steps
- A yellow caution sign for steps that must be completed before submitting your design to Altera for HardCopy development
- An information callout for items you must verify

Selecting an item within the HardCopy II flow menu provides a description of the task and recommended action. The view in the HardCopy II Advisor differs depending on the device you select.

Figure 4–14 shows the HardCopy II Advisor with the Stratix II device selected.

Figure 4–14. HardCopy II Advisor with Stratix II Selected

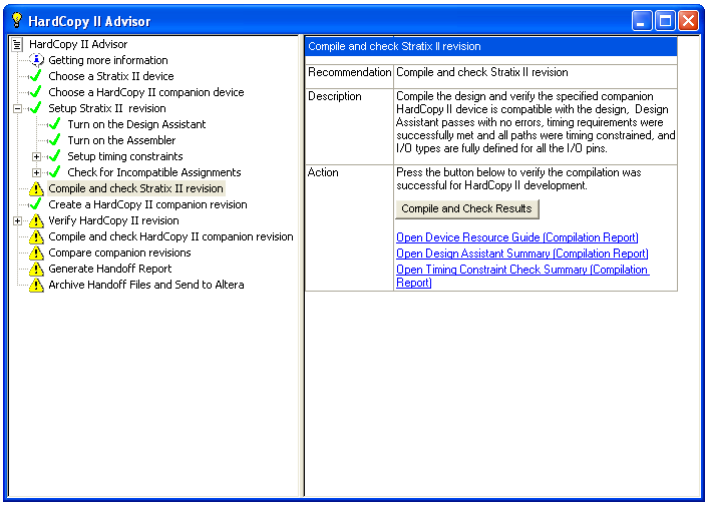
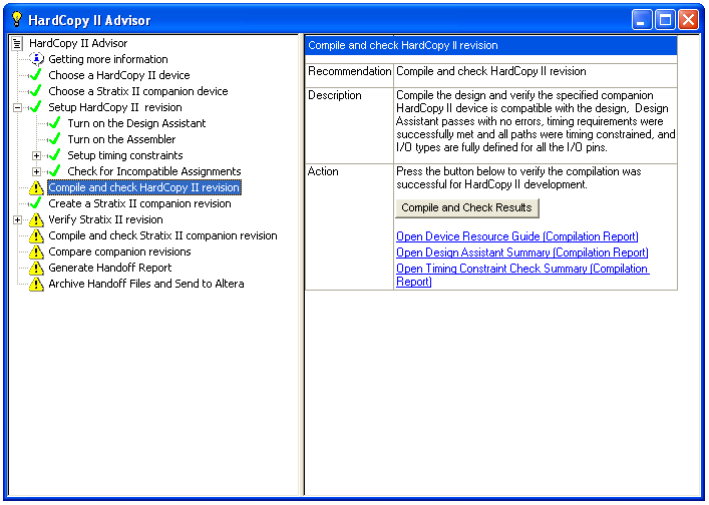


Figure 4–15 shows the HardCopy II Advisor with the HardCopy II device selected.

Figure 4–15. HardCopy II Advisor with HardCopy II Device Selected

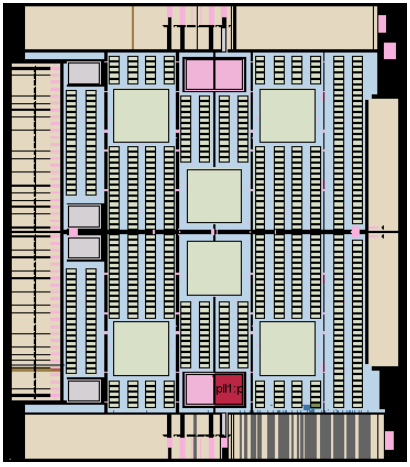


HardCopy II Floorplan View

The Quartus II software displays the preliminary timing closure floorplan and placement of your HardCopy II companion revision. This floorplan shows the preliminary placement and connectivity of all I/O pins, PLLs, memory blocks, HCell macros, and DSP HCell macros. Congestion mapping of routing connections can be viewed using the **Layers Setting** dialog box (in the View menu) settings. This is useful in analyzing densely packed areas of your floorplan that could be reducing the peak performance of your design. The HardCopy Design Center verifies final HCell macro timing and placement to guarantee timing closure is achieved.

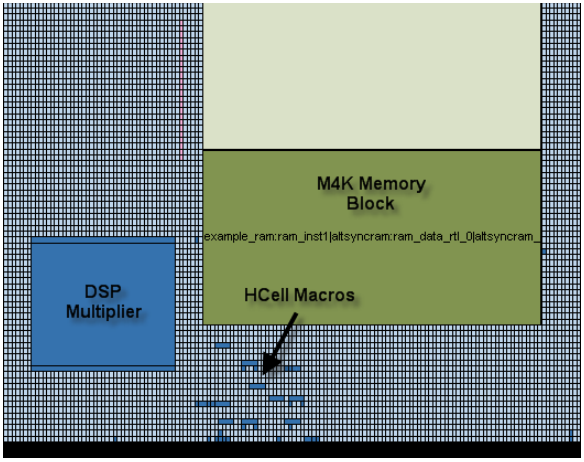
Figure 4–16 shows an example of the HC230F1020 device floorplan.

Figure 4–16. HC230F1020 Device Floorplan



In this small example design, the logic is placed near the bottom edge. You can see the placement of a DSP block constructed of HCell Macros, various logic HCell Macros, and an M4K memory block. A labeled close-up view of this region is shown in Figure 4–17.

Figure 4–17. Close-Up View of Floorplan



The HardCopy Design Center performs final placement and timing closure on your HardCopy II design based on the timing constraints provided in the Stratix II design.

For more information about the HardCopy Design Center's process, refer to the *Back-End Design Flow for HardCopy Series Devices* chapter in volume 1 of the *HardCopy Series Device Handbook*.

Conclusion

You can use the Quartus II software to design HardCopy II devices and to develop prototypes using Stratix II FPGAs. This is done using the standard FPGA development process with the addition of the HardCopy II Device Resource Guide, HardCopy II Companion Devices assignment HardCopy II Utilities, and the HardCopy II Advisor.

The addition of the HardCopy II Advisor to the Quartus II software provides an instrumental development guide for you to complete your HardCopy II and Stratix II device designs. The HardCopy II Utilities included in the Quartus II software provide you with the tools necessary to complete your Stratix II FPGA prototype and HardCopy II structured ASIC design. The addition of the HardCopy II companion revisions feature to the process allows for rapid development and verification that your HardCopy II design is functionally equivalent to your Stratix II FPGA prototype.

HardCopy Stratix Device Support

The Altera HardCopy devices provide a comprehensive alternative to ASICs. HardCopy structured ASICs offer a complete solution from prototype to high-volume production, and maintain the powerful features and high-performance architecture of their equivalent FPGAs with the programmability removed. You can use the Quartus II design software to design HardCopy devices in a manner similar to the traditional ASIC design flow and you can prototype with Altera's high density Stratix, APEX 20KC, and APEX 20KE FPGAs before seamlessly migrating to the corresponding HardCopy device for high-volume production.

HardCopy structured ASICs provide the following key benefits:

- Improves performance, on the average, by 40% over the corresponding -6 speed grade FPGA device
- Lowers power consumption, on the average, by 40% over the corresponding FPGA
- Preserves the FPGA architecture and features, and minimizes risk
- Guarantees first-silicon success through a proven, seamless migration process from the FPGA to the equivalent HardCopy device
- Offers a quick turnaround of the FPGA design to a structured ASIC device—samples are available in about eight weeks

Altera's Quartus II software has built-in support for HardCopy Stratix devices. The HardCopy design flow in Quartus II software offers the following advantages:

- Unified design flow from prototype to production
- Performance estimation of the HardCopy Stratix device allows you to design systems for maximum throughput
- Easy-to-use and inexpensive design tools from a single vendor
- An integrated design methodology that enables system-on-a-chip designs

This section discusses the following areas:

- How to design HardCopy Stratix and HardCopy APEX structured ASICs using the Quartus II software
- An explanation of what the HARDCOPY_FPGA_PROTOTYPE devices are and how to target designs to these devices
- Performance and power estimation of HardCopy Stratix devices
- How to generate the HardCopy design database for submitting HardCopy Stratix and HardCopy APEX designs to the HardCopy Design Center

Features

Beginning with version 4.2, the Quartus II software contains several powerful features that facilitate design of HardCopy Stratix and HardCopy APEX devices:

- **HARDCOPY_FPGA_PROTOTYPE Devices**
  These are virtual Stratix FPGA devices with features identical to HardCopy Stratix devices. You must use these FPGA devices to prototype your designs and verify the functionality in silicon.

- **HardCopy Timing Optimization Wizard**
  Using this feature, you can target your design to HardCopy Stratix devices, providing an estimate of the design's performance in a HardCopy Stratix device.

- **HardCopy Stratix Floorplans and Timing Models**
  The Quartus II software supports post-migration HardCopy Stratix device floorplans and timing models and facilitates design optimization for design performance.

- **Placement Constraints**
  Location and LogicLock™ constraints are supported at the HardCopy Stratix floorplan level to improve overall performance.

- **Improved Timing Estimation**
  Beginning with version 4.2, the Quartus II software determines routing and associated buffer insertion for HardCopy Stratix designs, and provides the Timing Analyzer with more accurate information about the delays than was possible in previous versions of the Quartus II software. The Quartus II Archive File automatically receives buffer insertion information, which greatly enhances the timing closure process in the back-end migration of your HardCopy Stratix device.

- **Design Assistant**
  This feature checks your design for compliance with all HardCopy device design rules and establishes a seamless migration path in the quickest time.

- **HardCopy Files Wizard**
  This wizard enables you to deliver to Altera the design database and all the deliverables required for migration. This feature is used for HardCopy Stratix and HardCopy APEX devices.

The HardCopy Stratix and HardCopy APEX PowerPlay Early Power Estimator is available on the Altera web site at www.altera.com.

HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix & Stratix Devices

You must use the HARDCOPY_FPGA_PROTOTYPE virtual devices available in the Quartus II software to target your designs to the actual resources and package options available in the equivalent post-migration HardCopy Stratix device. The programming file generated for the HARDCOPY_FPGA_PROTOTYPE can be used in the corresponding Stratix FPGA device.

The purpose of the HARDCOPY_FPGA_PROTOTYPE is to guarantee seamless migration to HardCopy by making sure that your design only uses resources in the FPGA that can be used in the HardCopy device after migration. You can use the equivalent Stratix FPGAs to verify the design's functionality in-system, then generate the design database necessary to migrate to a HardCopy device. This process ensures the seamless migration of the design from a prototyping device to a production device in high volume. It also minimizes risk, assures samples in about eight weeks, and guarantees first-silicon success.

HARDCOPY_FPGA_PROTOTYPE devices are only available for HardCopy Stratix devices and are not available for the HardCopy II or HardCopy APEX device families.

Table 4–4 compares HARDCOPY_FPGA_PROTOTYPE devices, Stratix devices, and HardCopy Stratix devices.

Table 4–4. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE to Stratix & HardCopy Stratix Devices

Stratix Device	HARDCOPY_FPGA_PROTOTYPE Device	HardCopy Stratix Device
FPGA	Virtual FPGA	Structured ASIC
FPGA	Architecture identical to Stratix FPGA	Architecture identical to Stratix FPGA
FPGA	Resources identical to HardCopy Stratix device	M-RAM resources different than Stratix FPGA in some devices
Ordered through Altera part number	Cannot be ordered, use the Altera Stratix FPGA part number	Ordered by Altera part number

Table 4–5 lists the resources available in each of the HardCopy Stratix devices.

Table 4–5. HardCopy Stratix Device Physical Resources

Device	LEs	ASIC Equivalent Gates (K) *(1)*	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Maximum User I/O Pins
HC1S25F672	25,660	250	224	138	2	10	6	473
HC1S30F780	32,470	325	295	171	2 *(2)*	12	6	597
HC1S40F780	41,250	410	384	183	2 *(2)*	14	6	615
HC1S60F1020	57,120	570	574	292	6	18	12	773
HC1S80F1020	79,040	800	767	364	6 *(2)*	22	12	773

Notes to Table 4–5:
(1) Combinational and registered logic do not include DSP blocks, on-chip RAM, or PLLs.
(2) The M-RAM resources for these HardCopy devices differ from the corresponding Stratix FPGA.

For a given device, the number of available M-RAM blocks in HardCopy Stratix devices is identical with the corresponding HARDCOPY_FPGA_PROTOTYPE devices, but may be different from the corresponding Stratix devices. Maintaining the identical resources between HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix devices facilitates seamless migration from the FPGA to the structured ASIC device.

For more information about HardCopy Stratix devices, refer to the *HardCopy Stratix Device Family Data Sheet* section in volume 1 of the *HardCopy Series Handbook.*

The three devices, Stratix FPGA, HARDCOPY_FPGA_PROTOTYPE, and HardCopy device, are distinct devices in the Quartus II software. The HARDCOPY_FPGA_PROTOTYPE programming files are used in the Stratix FPGA for your design. The three devices are tied together with the same netlist, thus a single SRAM Object File (**.sof)** can be used to achieve the various goals at each stage. The same SRAM Object File is generated in the HARDCOPY_FPGA_PROTOTYPE design, and is used to program the Stratix FPGA device, the same way that it is used to generate the HardCopy Stratix device, guaranteeing a seamless migration.

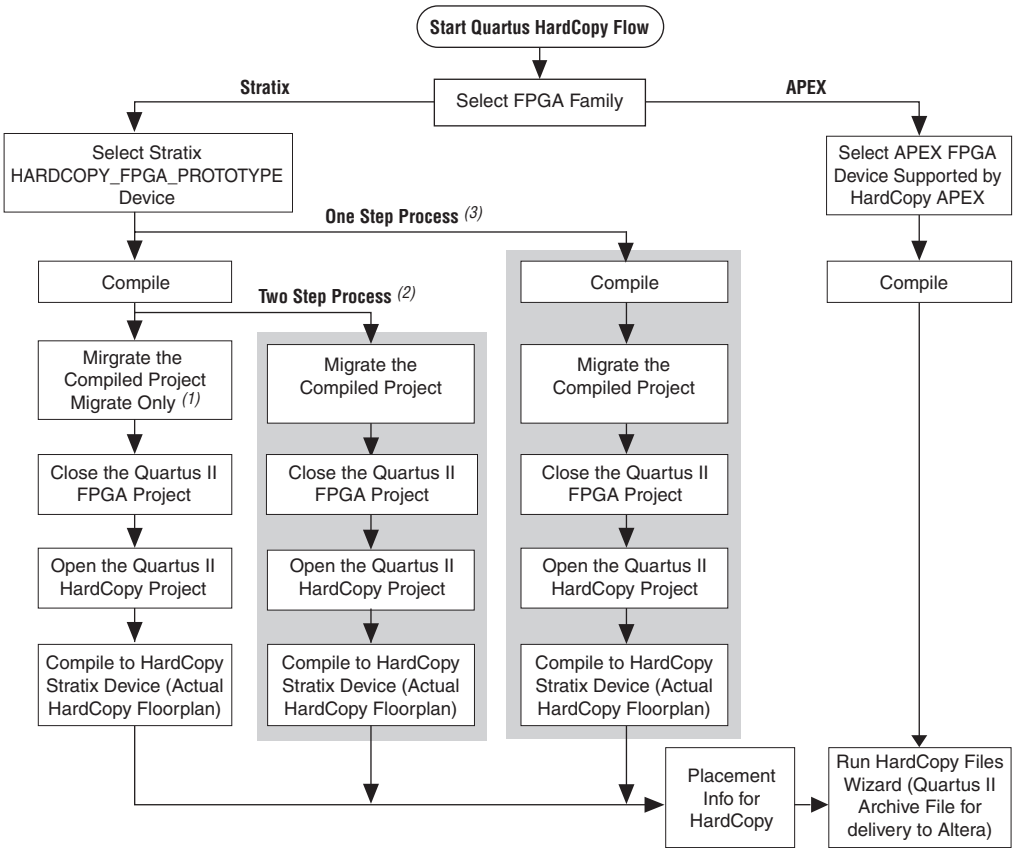
For more information about the SRAM Object File and programming Stratix FPGA devices, refer to the *Programming and Configuration* chapter of the *Introduction to Quartus II Manual*.

HardCopy Design Flow

Figure 4–18 shows a HardCopy design flow diagram. The design steps are explained in detail in the following sections of this chapter. The HardCopy Stratix design flow utilizes the HardCopy Timing Optimization Wizard to automate the migration process into a one-step process. The remainder of this section explains the tasks performed by this automated process.

For a detailed description of the HardCopy Timing Optimization Wizard and HardCopy Files Wizard, refer to "HardCopy Timing Optimization Wizard" on page 4–46 and "Generating the HardCopy Design Database" on page 4–57.

Figure 4–18. HardCopy Stratix & HardCopy APEX Design Flow Diagram



Notes for Figure 4–18:
(1) Migrate-Only Process: The displayed flow is completed manually.
(2) Two-Step Process: Migration and Compilation are done automatically (shaded area).
(3) One-Step Process: Full HardCopy Compilation. The entire process is completed automatically (shaded area).

The Design Flow Steps of the One Step Process

The following sections describe each step of the full HardCopy compilation (the One Step Process), as shown in Figure 4–18.

Compile the Design for an FPGA

This step compiles the design for a HARDCOPY_FPGA_PROTOTYPE device and gives you the resource utilization and performance of the FPGA.

Migrate the Compiled Project

This step generates the Quartus II Project File (**.qpf**) and the other files required for HardCopy implementation. The Quartus II software also assigns the appropriate HardCopy Stratix device for the design migration.

Close the Quartus FPGA Project

Because you must compile the project for a HardCopy Stratix device, you must close the existing project which you have targeted your design to a HARDCOPY_FPGA_PROTOTYPE device.

Open the Quartus HardCopy Project

Open the Quartus II project that you created in the "Migrate the Compiled Project" step. The selected device is one of the devices from the HardCopy Stratix family that was assigned during that step.

Compile for HardCopy Stratix Device

Compile the design for a HardCopy Stratix device. After successful compilation, the Timing Analysis section of the compilation report shows the performance of the design implemented in the HardCopy device.

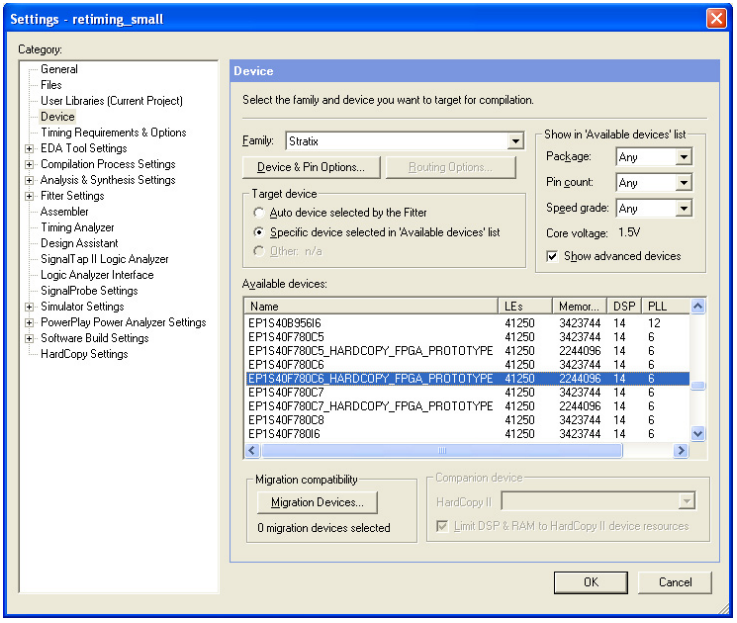
How to Design HardCopy Stratix Devices

This section describes the process for designing for a HardCopy Stratix device using the HARDCOPY_FPGA_PROTOTYPE as your initial selected device. In order to use the HardCopy Timing Optimization Wizard, you must first design with the HARDCOPY_FPGA_PROTOTYPE in order for the design to migrate to a HardCopy Stratix device.

To target a design to a HardCopy Stratix device in the Quartus II software, follow these steps:

1. If you have not yet done so, create a new project or open an existing project.
2. On the Assignments menu, click **Settings**. In the **Category** list, select **Device**.
3. On the **Device** page, in the **Family** list, select **Stratix**. Select the desired HARDCOPY_FPGA_PROTOTYPE device in the **Available Devices** list (Figure 4–19).

Figure 4–19. Selecting a HARDCOPY_FPGA_PROTOTYPE Device



By choosing the HARDCOPY_FPGA_PROTOTYPE device, all the design information, available resources, package option, and pin assignments are constrained to guarantee a seamless migration of your project to the HardCopy Stratix device. The netlist resulting from the HARDCOPY_FPGA_PROTOTYPE device compilation contains information about the electrical connectivity, resources used, I/O placements, and the unused resources in the FPGA device.

4. On the Assignments menu, click **Settings**. In the **Category** list, select **HardCopy Settings** and specify the input transition timing to be modeled for both clock and data input pins. These transition times are used in static timing analysis during back-end timing closure of the HardCopy device.

5. Add constraints to your HARDCOPY_FPGA_PROTOTYPE device, and on the Processing menu, click **Start Compilation** to compile the design.

HardCopy Timing Optimization Wizard

After you have successfully compiled your design in the HARDCOPY_FPGA_PROTOTYPE, you must migrate the design to the HardCopy Stratix device to get a performance estimation of the HardCopy Stratix device. This migration is required before submitting the design to Altera for the HardCopy Stratix device implementation. To perform the required migration, on the Project menu, point to HardCopy Utilities and click **HardCopy Timing Optimization Wizard**.

At this point, you are presented with the following three choices to target the designs to HardCopy Stratix devices (Figure 4–20):

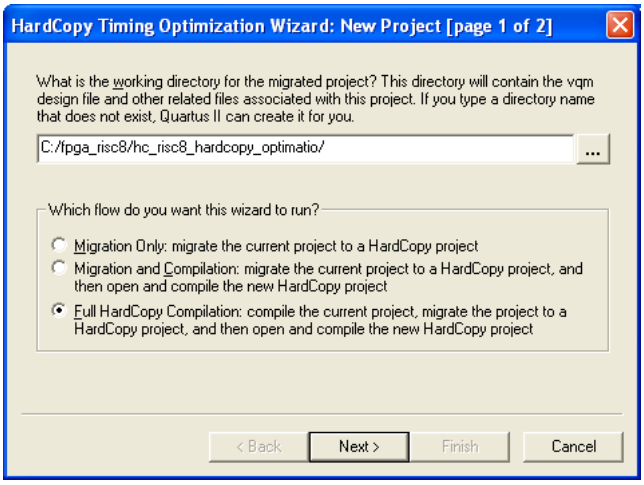
- **Migration Only**: You can select this option after compiling the HARDCOPY_FPGA_PROTOTYPE project to migrate the project to a HardCopy Stratix project.

  You can now perform the following tasks manually to target the design to a HardCopy Stratix device. Refer to"Performance Estimation" on page 4–49 for additional information about how to perform these tasks.
  - Close the existing project
  - Open the migrated HardCopy Stratix project
  - Compile the HardCopy Stratix project for a HardCopy Stratix device

- **Migration and Compilation**: You can select this option after compiling the project. This option results in the following actions:
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling the project for a HardCopy Stratix device

- **Full HardCopy Compilation**: Selecting this option results in the following actions:
  - Compiling the existing HARDCOPY_FPGA_PROTOTYPE project
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling it for a HardCopy Stratix device

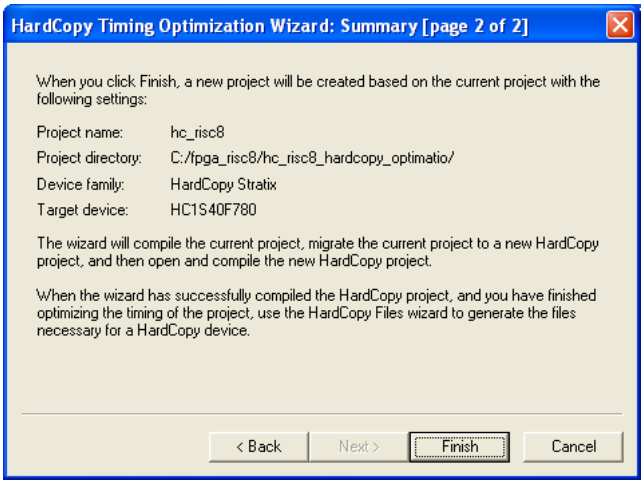
Figure 4–20. HardCopy Timing Optimization Wizard Options



The main benefit of the HardCopy Timing Wizard's three options is flexibility of the conversion process automation. The first time you migrate your HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix device, you may want to use Migration Only, and then work on the HardCopy Stratix project in the Quartus II software. As your prototype FPGA project and HardCopy Stratix project constraints stabilize and you have fewer changes, the Full HardCopy Compilation is ideal for one-click compiling of your HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix projects.

After selecting the wizard you want to run, the "HardCopy Timing Optimization Wizard: Summary" page shows you details about the settings you made in the Wizard, as shown in Figure 4–21.

Figure 4–21. HardCopy Timing Optimization Wizard Summary Page



When either of the second two options in Figure 4–20 are selected (**Migration and Compilation** or **Full HardCopy Compilation**), designs are targeted to HardCopy Stratix devices and optimized using the HardCopy Stratix placement and timing analysis to estimate performance. For details on the performance optimization and estimation steps, refer to "Performance Estimation" on page 4–49. If the performance requirement is not met, you can modify your RTL source, optimize the FPGA design, and estimate timing until you reach timing closure.

Tcl Support for HardCopy Migration

To complement the GUI features for HardCopy migration, the Quartus II software provides the following command-line executables (which provide the tool command language (Tcl) shell to run the `--flow` Tcl command) to migrate the HARDCOPY_FPGA_PROTOTYPE project to HardCopy Stratix devices:

```
quartus_sh --flow migrate_to_hardcopy <project_name> [-c <revision>] ↵
```

This command migrates the project compiled for the HARDCOPY_FPGA_PROTOTYPE device to a HardCopy Stratix device.

```
quartus_sh --flow hardcopy_full_compile <project_name> [-c <revision>] ↵
```

This command performs the following tasks:

- Compiles the exsisting project for a HARDCOPY_FPGA_PROTOTYPE device.
- Migrates the project to a HardCopy Stratix project.
- Opens the migrated HardCopy Stratix project and compiles it for a HardCopy Stratix device.

Design Optimization & Performance Estimation

The HardCopy Timing Optimization Wizard creates the HardCopy Stratix project in the Quartus II software, where you can perform design optimization and performance estimation of your HardCopy Stratix device.

Design Optimization

Beginning with version 4.2, the Quartus II software supports HardCopy Stratix design optimization by providing floorplans for placement optimization and HardCopy Stratix timing models. These features enable you to refine placement of logic array blocks (LAB) and optimize the HardCopy design further than the FPGA performance. Customized routing and buffer insertion done in the Quartus II software are then used to estimate the design's performance in the migrated device. The HardCopy device floorplan, routing, and timing estimates in the Quartus II software reflect the actual placement of the design in the HardCopy Stratix device, and can be used to see the available resources, and the location of the resources in the actual device.

Performance Estimation

Figure 4–22 illustrates the design flow for estimating performance and optimizing your design. You can target your designs to HARDCOPY_FPGA_PROTOTYPE devices, migrate the design to the HardCopy Stratix device, and get placement optimization and timing estimation of your HardCopy Stratix device.

In the event that the required performance is not met, you can:

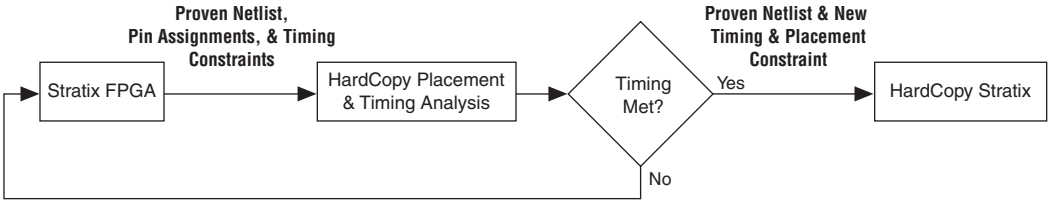
- Work to improve LAB placement in the HardCopy Stratix project.

  or

- Go back to the HARDCOPY_FPGA_PROTOTYPE project and optimize that design, modify your RTL source code, repeat the migration to the HardCopy Stratix device, and perform the optimization and timing estimation steps.

On average, HardCopy Stratix devices are 40% faster than the equivalent -6 speed grade Stratix FPGA device. These performance numbers are highly design dependent, and you must obtain final performance numbers from Altera.

Figure 4–22. Obtaining a HardCopy Performance Estimation



To perform Timing Analysis for a HardCopy Stratix device, follow these steps:

1. Open an existing project compiled for a HARDCOPY_FPGA_PROTOYPE device.

2. On the Project menu, point to HardCopy Utilities and click **HardCopy Timing Optimization Wizard**.

3. Select a destination directory for the migrated project and complete the HardCopy Timing Optimization Wizard process.

   On completion of the HardCopy Timing Optimization Wizard, the destination directory created contains the Quartus II project file, and all files required for HardCopy Stratix implementation. At this stage, the design is copied from the HARDCOPY_FPGA_PROTOTYPE project directory to a new directory to perform the timing analysis. This two-project directory structure enables you to move back and forth between the HARDCOPY_FPGA_PROTOTYPE design database and the HardCopy Stratix design database. The Quartus II software creates the <*project name*>**_hardcopy_optimization** directory.

   You do not have to select the HardCopy Stratix device while performing performance estimation. When you run the HardCopy Timing Optimization Wizard, the Quartus II software selects the HardCopy Stratix device corresponding to the specified HARDCOPY_FPGA_PROTOTYPE FPGA. Thus, the information necessary for the HardCopy Stratix device is available from the earlier HARDCOPY_FPGA_PROTOTYPE device selection.

All constraints related to the design are also transferred to the new project directory. You can modify these constraints, if necessary, in your optimized design environment to achieve the necessary timing closure. However, if the design is optimized at the HARDCOPY_FPGA_PROTOTYPE device level by modifying the RTL code or the device constraints, you must migrate the project with the HardCopy Timing Optimization Wizard.

If an existing project directory is selected when the HardCopy Timing Optimization Wizard is run, the existing information is overwritten with the new compile results.

The project directory is the directory that you chose for the migrated project. A snapshot of the files inside the <*project name*>**_hardcopy_optimization** directory is shown in Table 4–6.

Table 4–6. Directory Structure Generated by the HardCopy Timing Optimization Wizard

<*project name*>**_hardcopy_optimization**
- <*project name*>**.qsf**
- <*project name*>**.qpf**
- <*project name*>**.sof**
- <*project name*>**.macr**
- <*project name*>**.gclk**
- **db**
- **hardcopy_fpga_prototype**
  - fpga_<*project name*>**_violations.datasheet**
  - fpga_<*project name*>**_target.datasheet**
  - fpga_<*project name*>**_rba_pt_hcpy_v.tcl**
  - fpga_<*project name*>**_pt_hcpy_v.tcl**
  - fpga_<*project name*>**_hcpy_v.sdo**
  - fpga_<*project name*>**_hcpy.vo**
  - fpga_<*project name*>**_cpld.datasheet**
  - fpga_<*project name*>**_cksum.datasheet**
  - fpga_<*project name*>**.tan.rpt**
  - fpga_<*project name*>**.map.rpt**
  - fpga_<*project name*>**.map.atm**
  - fpga_<*project name*>**.fit.rpt**
  - fpga_<*project name*>**.db_info**
  - fpga_<*project name*>**.cmp.xml**
  - fpga_<*project name*>**.cmp.rcf**
  - fpga_<*project name*>**.cmp.atm**
  - fpga_<*project name*>**.asm.rpt**
  - fpga_<*project name*>**.qarlog**
  - fpga_<*project name*>**.qar**
  - fpga_<*project name*>**.qsf**
  - fpga_<*project name*>**.pin**
  - fpga_<*project name*>**.qpf**
- **db_export**
  - <*project name*>**.map.atm**
  - <*project name*>**.map.hdbx**
  - <*project name*>**.db_info**

4. Open the migrated Quartus II project created in Step 3.

5. Perform a full compilation.

   After successful compilation, the Timing Analysis section of the Compilation Report shows the performance of the design.

Performance estimation is not supported for HardCopy APEX devices in the Quartus II software. Your design can be optimized by modifying the RTL code or the FPGA design and the constraints. You should contact Altera to discuss any desired performance improvements with HardCopy APEX devices.

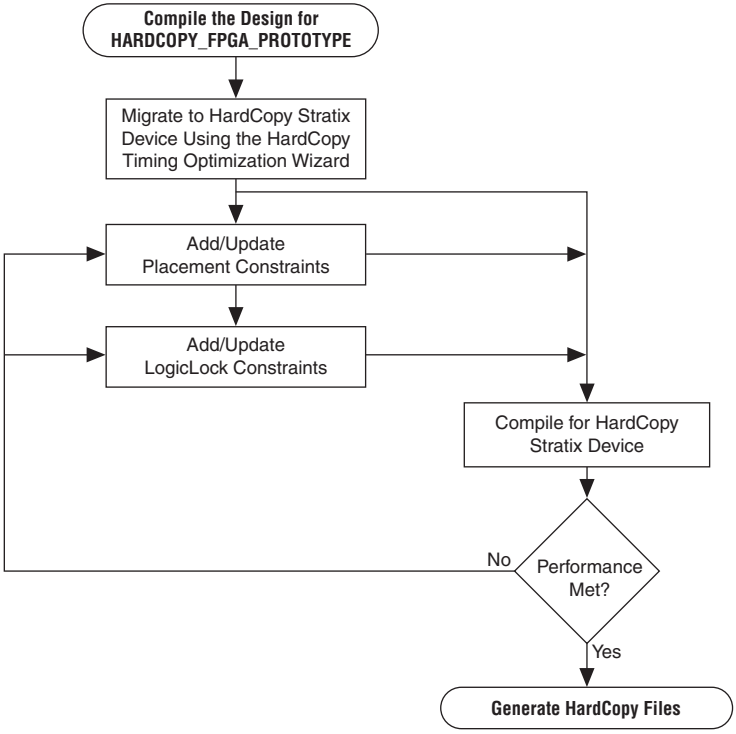
Buffer Insertion

Beginning with version 4.2, the Quartus II software provides improved HardCopy Stratix device timing closure and estimation, to more accurately reflect the results expected after back-end migration. The Quartus II software performs the necessary buffer insertion in your HardCopy Stratix device during the Fitter process, and stores the location of these buffers and necessary routing information in the Quartus II Archive File. This buffer insertion improves the estimation of the Quartus II Timing Analyzer for the HardCopy Stratix device.

Placement Constraints

Beginning with version 4.2, the Quartus II software supports placement constraints and LogicLock regions for HardCopy Stratix devices. Figure 4–23 shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is achieved.

Figure 4–23. Placement Constraints Flow for HardCopy Stratix Devices



Location Constraints

This section provides information about HardCopy Stratix logic location constraints.

LAB Assignments

Logic placement in HardCopy Stratix is limited to LAB placement and optimization of the interconnecting signals between them. In a Stratix FPGA, individual logic elements (LE) are placed by the Quartus II Fitter into LABs. The HardCopy Stratix migration process requires that LAB contents cannot change after the Timing Optimization Wizard task is done. Therefore you can only make LAB-level placement optimization and location assignments after migrating the HARDCOPY_FPGA_PROTOTYPE project to the HardCopy Stratix device.

The Quartus II software supports these LAB location constraints for HardCopy Stratix devices. The entire contents of a LAB is moved to an empty LAB when using LAB location assignments. If you want to move the logic contents of LAB A to LAB B, the entire contents of LAB A are moved to an empty LAB B. For example, the logic contents of LAB_X33_Y65 can be moved to an empty LAB at LAB_X43_Y56 but individual logic cell LC_X33_Y65_N1 can not be moved by itself in the HardCopy Stratix Timing Closure Floorplan.

LogicLock Assignments

The LogicLock feature of the Quartus II software provides a block-based design approach. Using this technique you can partition your design and create each block of logic independently, optimize placement and area, and integrate all blocks into the top level design.

To learn more about this methodology, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

LogicLock constraints are supported when you migrate the project from a HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix project. If the LogicLock region was specified as "Size=Fixed" and "Location=Locked" in the HARDCOPY_FPGA_PROTOTYPE project, it is converted to have "Size=Auto" and "Location=Floating" as shown in the following LogicLock examples. This modification is necessary because the floorplan of a HardCopy Stratix device is different from that of the Stratix device, and the assigned coordinates in the HARDCOPY_FPGA_PROTOTYPE do not match the HardCopy Stratix floorplan. If this modification did not occur, LogicLock assignments would lead to incorrect placement in the Quartus II Fitter. Making the regions auto-size and floating, maintains your LogicLock assignments, allowing you to easily adjust the LogicLock regions as required and lock their locations again after HardCopy Stratix placement.

Example 4–1 and Example 4–2 show two examples of LogicLock assignments.

Example 4–1. LogicLock Region Definition in the HARDCOPY_FPGA_PROTOTYPE Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE LOCKED -entity risc8 -section_id test
set_global_assignment -name LL_AUTO_SIZE OFF -entity risc8 -section_id test
```

Example 4–2. LogicLock Region Definition in the Migrated HardCopy Stratix Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE FLOATING -entity risc8 -section_id
test
set_global_assignment -name LL_AUTO_SIZE ON -entity risc8 -section_id test
```

Checking Designs for HardCopy Design Guidelines

When you develop a design with HardCopy migration in mind, you must follow Altera recommended design practices that ensure a straightforward migration process or the design will not be able to be implemented in a HardCopy device. Prior to starting migration of the design to a HardCopy device, you must review the design and identify and address all the design issues. Any design issues that have not been addressed can jeopardize silicon success.

Altera-Recommended HDL Coding Guidelines

Designing for Altera PLD, FPGA, and HardCopy structured ASIC devices requires certain specific design guidelines and hardware description language (HDL) coding style recommendations be followed.

For more information about design recommendations and HDL coding styles, refer to the *Design Guidelines* Section in volume 1 of the *Quartus II Handbook*.

Design Assistant

The Quartus II software includes the Design Assistant feature to check your design against the HardCopy design guidelines. Some of the design rule checks performed by the Design Assistant include the following rules:

- Design should not contain combinational loops
- Design should not contain delay chains
- Design should not contain latches

To use the Design Assistant, you must have at least run Analysis and Synthesis on the design in the Quartus II software. Altera recommends that you run the Design Assistant to check for compliance with the HardCopy design guidelines early in the design process and after every compilation.

Design Assistant Settings

You must select the design rules in the **Design Assistant** page prior to running the design. On the Assignments menu, click **Settings**. In the **Settings** dialog box, in the Category list, select **Design Assistant** and turn on **Run Design Assistant during compilation**. Altera recommends enabling this feature to run the Design Assistant automatically during compilation of your design.

Running Design Assistant

To run Design Assistant independently of other Quartus II features, on the Processing menu, point to Start and click **Start Design Assistant**.

The Design Assistant automatically runs in the background of the Quartus II software when the HardCopy Timing Optimization Wizard is launched, and does not display the Design Assistant results immediately to the display. The design is checked before the Quartus II software migrates the design and creates a new project directory for performing timing analysis.

Also, the Design Assistant runs automatically whenever you generate the HardCopy design database with the HardCopy Files Wizard. The Design Assistant report generated is used by the Altera HardCopy Design Center to review your design.

Reports & Summary

The results of running the Design Assistant on your design are available in the Design Assistant Results section of the Compilation Report. The Design Assistant also generates the summary report in the <*project name*>**hardcopy** subdirectory of the project directory. This report file is titled <*project name*>**_violations.datasheet**. Reports include the settings, run summary, results summary, and details of the results and messages. The Design Assistant report indicates the rule name, severity of the violation and the circuit path where any violation occurred.

To learn about the design rules and standard design practices to comply with HardCopy design rules, refer to the Quartus II Help and the *HardCopy Series Design Guidelines* chapter in volume 1 of the *HardCopy Series Handbook*.

Generating the HardCopy Design Database

You can use the HardCopy Files Wizard to generate the complete set of deliverables required for migrating the design to a HardCopy device in a single click. The HardCopy Files Wizard asks questions related to the design and archives your design, settings, results, and database files for delivery to Altera. Your responses to the design details are stored in *<project name>*_**hardcopy_optimization***<project name>*.**hps.txt**.

You can generate the archive of the HardCopy design database only after compiling the design to a HardCopy Stratix device. The Quartus II Archive File is generated at the same directory level as the targeted project, either before or after optimization.

The Design Assistant automatically runs when the HardCopy Files Wizard is started.

Table 4–7 shows the archive directory structure and files collected by the HardCopy Files Wizard.

Table 4–7. HardCopy Stratix Design Files Collected by the HardCopy Files Wizard

<*project name*>**_hardcopy_optimization**
- <*project name*>**.flow.rpt**
- <*project name*>**.qpf**
- <*project name*>**.asm.rpt**
- <*project name*>**.blf**
- <*project name*>**.fit.rpt**
- <*project name*>**.gclk**
- <*project name*>**.hps.txt**
- <*project name*>**.macr**
- <*project name*>**.pin**
- <*project name*>**.qsf**
- <*project name*>**.sof**
- <*project name*>**.tan.rpt**

hardcopy
- <*project name*>**.apc**
- <*project name*>**_cksum.datasheet**
- <*project name*>**_cpld.datasheet**
- <*project name*>**_hcpy.vo**
- <*project name*>**_hcpy_v.sdo**
- <*project name*>**_pt_hcpy_v.tcl**
- <*project name*>**_rba_pt_hcpy_v.tcl**
- <*project name*>**_target.datasheet**
- <*project name*>**_violations.datasheet**

hardcopy_fpga_prototype
- fpga_<*project name*>**.asm.rpt**
- fpga_<*project name*>**.cmp.rcf**
- fpga_<*project name*>**.cmp.xml**
- fpga_<*project name*>**.db_info**
- fpga_<*project name*>**.fit.rpt**
- fpga_<*project name*>**.map.atm**
- fpga_<*project name*>**.map.rpt**
- fpga_<*project name*>**.pin**
- fpga_<*project name*>**.qsf**
- fpga_<*project name*>**.tan.rpt**
- fpga_<*project name*>**_cksum.datasheet**
- fpga_<*project name*>**_cpld.datasheet**
- fpga_<*project name*>**_hcpy.vo**
- fpga_<*project name*>**_hcpy_v.sdo**
- fpga_<*project name*>**_pt_hcpy_v.tcl**
- fpga_<*project name*>**_rba_pt_hcpy_v.tcl**
- fpga_<*project name*>**_target.datasheet**
- fpga_<*project name*>**_violations.datasheet**

db_export
- <*project name*>**.db_info**
- <*project name*>**.map.atm**
- <*project name*>**.map.hdbx**

After creating the migration database with the HardCopy Timing Optimization Wizard, you must compile the design before generating the project archive. You will receive an error if you create the archive before compiling the design.

Static Timing Analysis

In addition to performing timing analysis, the Quartus II software also provides all of the requisite netlists and Tcl scripts to perform static timing analysis (STA) using the Synopsys STA tool, PrimeTime. The following files, necessary for timing analysis with the PrimeTime tool, are generated by the HardCopy Files Wizard:

- <*project name*>**_hcpy.vo**—Verilog HDL output format
- <*project name*>**_hpcy_v.sdo**—Standard Delay Format Output File
- <*project name*>**_pt_hcpy_v.tcl**—Tcl script

These files are available in the <*project name*>**\hardcopy** directory. PrimeTime libraries for the HardCopy Stratix and Stratix devices are included with the Quartus II software.

Use the HardCopy Stratix libraries for PrimeTime to perform STA during timing analysis of designs targeted to HARDCOPY_FPGA_PROTOTYPE device.

For more information about static timing analysis, refer to the *Classic Timing Analyzer* and the *Synopsys PrimeTime Support* chapters in volume 3 of the *Quartus II Handbook*.

Early Power Estimation

You can use PowerPlay Early Power Estimation to estimate the amount of power your HardCopy Stratix or HardCopy APEX device will consume. This tool is available on the Altera web site. Using the Early Power Estimator requires some knowledge of your design resources and specifications, including:

- Target device and package
- Clock networks used in the design
- Resource usage for LEs, DSP blocks, PLL, and RAM blocks
- High speed differential interfaces (HSDI), general I/O power consumption requirements, and pin counts
- Environmental and thermal conditions

HardCopy Stratix Early Power Estimation

The PowerPlay Early Power Estimator provides an initial estimate of I_{CC} for any HardCopy Stratix device based on typical conditions. This calculation saves significant time and effort in gaining a quick understanding of the power requirements for the device. No stimulus vectors are necessary for power estimation, which is established by the clock frequency and toggle rate in each clock domain.

This calculation should only be used as an estimation of power, not as a specification. The actual I_{CC} should be verified during operation because this estimate is sensitive to the actual logic in the device and the environmental operating conditions.

For more information about simulation-based power estimation, refer to the *Power Estimation & Analysis* Section in volume 3 of the *Quartus II Handbook*.

On average, HardCopy Stratix devices are expected to consume 40% less power than the equivalent FPGA.

HardCopy APEX Early Power Estimation

The PowerPlay Early Power Estimator can be run from the Altera web site in the device support section (**http://www.altera.com/support/devices/dvs-index.html**). You cannot open this feature in the Quartus II software.

With the HardCopy APEX PowerPlay Early Power Estimator, you can estimate the power consumed by HardCopy APEX devices and design systems with the appropriate power budget. Refer to the web page for instructions on using the HardCopy APEX PowerPlay Early Power Estimator.

HardCopy APEX devices are generally expected to consume about 40% less power than the equivalent APEX 20KE or APEX 20KC FPGA devices.

Tcl Support for HardCopy Stratix

The Quartus II software also supports the HardCopy Stratix design flow at the command prompt using Tcl scripts.

For details on Quartus II support for Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Targeting Designs to HardCopy APEX Devices

Beginning with version 4.2, the Quartus II software supports targeting designs to HardCopy APEX device families. After compiling your design for one of the APEX 20KC or APEX 20KE FPGA devices supported by a HardCopy APEX device, run the HardCopy Files Wizard to generate the necessary set of files for HardCopy migration.

The HardCopy APEX device requires a different set of design files for migration than HardCopy Stratix. Table 4–8 shows the files collected for HardCopy APEX by the HardCopy Files Wizard.

Table 4–8. HardCopy APEX Files Collected by the HardCopy Files Wizard

<*project name*>**.tan.rpt**
<*project name*>**.asm.rpt**
<*project name*>**.fit.rpt**
<*project name*>**.hps.txt**
<*project name*>**.map.rpt**
<*project name*>**.pin**
<*project name*>**.sof**
<*project name*>**.qsf**
<*project name*>**_cksum.datasheet**
<*project name*>**_cpld.datasheet**
<*project name*>**_hcpy.vo**
<*project name*>**_hcpy_v.sdo**
<*project name*>**_pt_hcpy_v.tcl**
<*project name*>**_rba_pt_hcpy_v.tcl**
<*project name*>**_target.datasheet**
<*project name*>**_violations.datasheet**

Refer to "Generating the HardCopy Design Database" on page 4–57 for information about generating the complete set of deliverables required for migrating the design to a HardCopy APEX device. After you have successfully run the HardCopy Files Wizard, you can submit your design archive to Altera to implement of your design in a HardCopy device. You should contact Altera for more information about this process.

Conclusion

The methodology for designing HardCopy Stratix devices using the Quartus II software is the same as that for designing the Stratix FPGA equivalent. You can use the familiar Quartus II software tools and design flow, target designs to HardCopy Stratix devices, optimize designs for higher performance and lower power consumption than the Stratix FPGAs, and deliver the design database for migration to a HardCopy Stratix device. Compatible APEX FPGA designs can migrate to HardCopy APEX after compilation using the HardCopy Files Wizard to archive the design files. Submit the files to the HardCopy Design Center to complete the back-end migration.

Related Documents

For more information, refer to the following documentation:

- The *HardCopy Series Design Guidelines* chapter in volume 1 of the *HardCopy Series Handbook*
- The *HardCopy Series Back-End Timing Closure* chapter in volume 1 of the *HardCopy Series Handbook*

Document Revision History

Table 4–9 shows the revision history for this chapter.

Table 4–9. Documentation Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Minor updates for the Quartus II software version 6.1.0 • Added "Performing ECOs with Change Manager and Chip Planner" and "Overall Migration Flow" sections. • Updated "Quartus II Software Features Supported for HardCopy II Designs" section.	A medium update to the chapter, due to changes in the Quartus II software version 6.1 release; most changes were in the "Performing ECOs with Change Manager and Chip Planner" and "Overall Migration Flow" sections.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	
May 2005 v5.0.0	• Chapter 3 was formerly Chapter 2. • Updated for consistency with the *Quartus II Support for HardCopy II Devices* and *Quartus II Support for HardCopy Stratix Devices* chapters in the *HardCopy Series Handbook*.	
Jan. 2005 v2.1	• Added HardCopy II Device Material.	
Dec. 2004 v2.1	• Chapter 2 was formerly Chapter 3. • Updates to tables, figures. • New functionality for Quartus II software 4.2	
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	
Feb. 2004 v1.0	Initial release.	

5. Engineering Change Management

QII51005-6.1.0

Introduction

Programmable logic can accommodate changes to a system specification late in the design cycle. In a typical engineering project development cycle, the specification for the programmable logic portion is likely to change when engineering development begins or while integrating all system elements.

Last-minute design changes, commonly referred to as engineering change orders (ECOs), are defined as small changes to the functionality of a design after the design has been fully compiled; that is, when synthesis and place-and-route are completed.

ECOs usually correct errors found in a design during debugging, or facilitate changes that are made to the design specification to compensate for design problems that may be introduced while integrating other components of the system design. As a project nears completion, a significant amount of time has likely been invested in maximizing performance and design verification; therefore, it is important that the ECO changes affect specific parts of the design and have minimal impact on other, unrelated parts of the design.

This chapter addresses the impact that ECOs have on the design cycle, explains the tools available in the Quartus® II software to perform ECOs, and offers solutions about how to resolve some of the issues related to ECO flow.

The Impact of Last-Minute Design Changes

ECOs have an impact on the following areas of a system design:

- Performance
- Compilation Time
- Verification
- Documentation

Performance

Making even a small change to the design functionality can result in a loss of previous design optimizations. Typical examples of design optimizations are floorplan optimizations and physical synthesis. Ideally, previous design optimizations should be preserved. This would focus future optimizations on design areas where the ECO changes occurred.

The Quartus II software offers an incremental compilation feature that preserves the optimizations and placement of your design while recompiling your designs. This feature allows you to create partitions of your design, so that if a change is required after the design is fully placed and optimized, only the affected partition is recompiled to implement the change. Beginning with the Quartus II software version 6.1, ECOs can be preserved during recompilation of the design partitions.

For more information about how to perform ECOs in your design, refer to the *Design Analysis & Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*.

For more information about how to use the incremental compilation feature in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical & 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; that is, Synthesis and Place-and-Route. Making small changes to the design to reach the final implementation on a board can be a very long process. Ideally, to reach the desired functionality and timing closure, a small change in functionality should result in reduced compilation time. You can achieve this by using the incremental compilation feature, which uses the previous fit information on unchanged areas of the design.

Verification

After a design change, you must verify the impact of the change on your design. You can verify your design by performing static timing analysis and simulation of your design. You can limit the verification to the area of the design affected by the ECOs. To do this for static timing analyses, run timing analysis on selected paths. To do this in simulation, perform the simulation on gate level and timing simulation netlists for the required partitions.

Documentation

You must track changes to your project files. Tracking changes provides you with the ability to reproduce your results. Ideally, you can have multiple compilation revisions so others can try the changes without corrupting the previous results.

Tools to Perform ECOs

The Quartus II software has powerful tools to make ECOs at any stage of your design.

The following tools are available in the Quartus II software to perform ECOs:

- Chip Planner
- Resource Property Editor
- Change Manager

For more information about how to use these tools to perform ECOs and design analysis, refer to the *Design Analysis & Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*.

ECO Support

You can apply ECOs at two stages in a typical design flow:

- ECO Support at the HDL Level
- ECO Support at the Netlist Level

ECO Support at the HDL Level

An ECO at the HDL level is a change to the design's Verilog HDL or VHDL source. This change may range from a single line to several lines of code modified within a module or entity. Typical examples of such modifications include:

- Changing the state encoding of a finite state machine
- Adding pipeline registers to improve design performance
- Duplicating the signal to reduce fan-out
- Adding a term to a conditional expression
- Changing the polarity of a register control signal

Figure 5–1 on page 5–5 shows the recommended design flow to support ECO changes at the HDL level. A few changes to the source code can produce many changes to the netlist produced by other EDA synthesis or tools such as the Quartus II software's integrated synthesis. During the synthesis process, the synthesis tools generally preserve the names of registers from the HDL source code, but automatically generate names

for the combinational (look-up table level, or LUT level) nodes. This automatic name generation is necessary to accommodate the synthesis optimization performed on the HDL source to use the target device resources more efficiently.

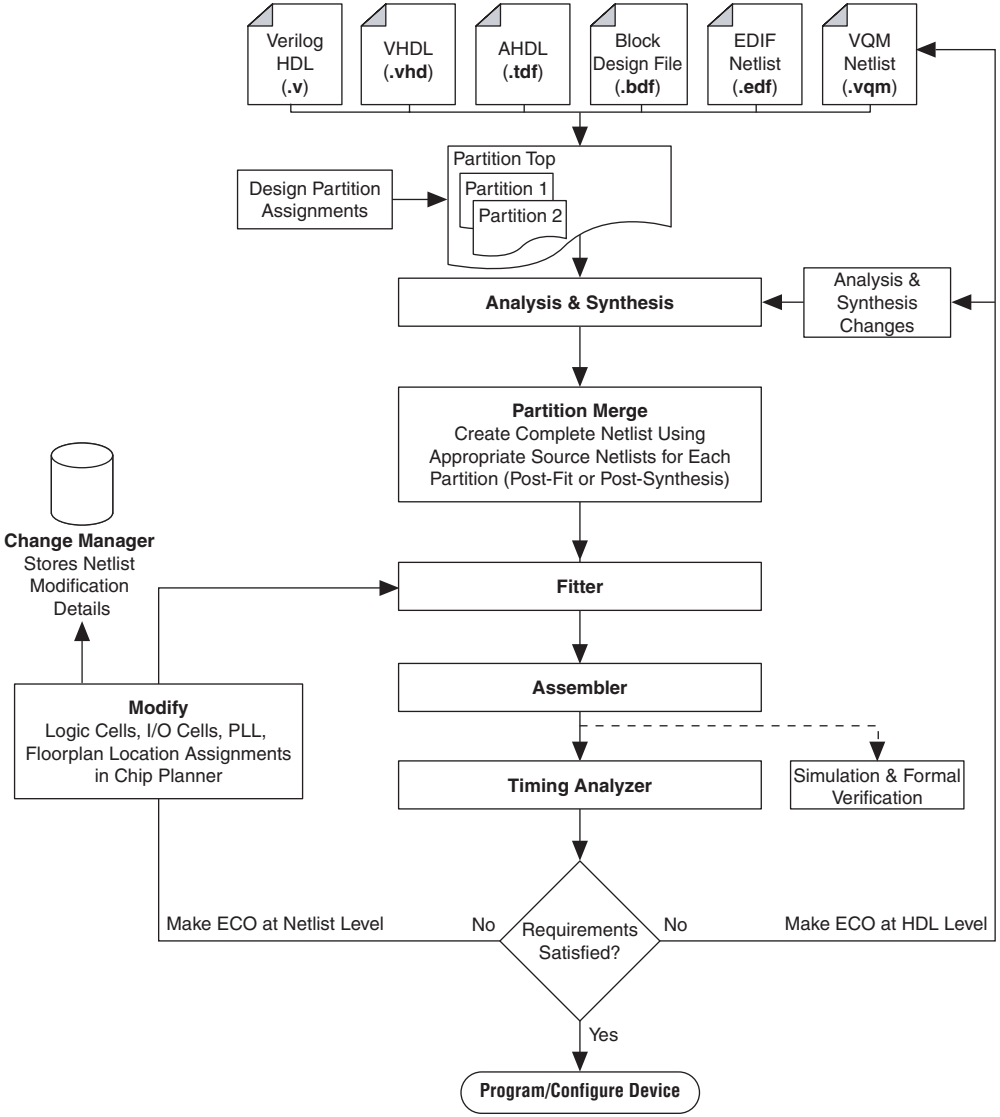
A minor source code change can result in many changes to the names in the synthesis netlist. The changes in the synthesis netlist can be caused by the node names in the new netlist that implements a different functionality than in the previous netlist. These changes can also be caused by implementing the same functionality as in the previous netlist, but using different names.

To leverage previous design optimizations and to reduce compilation time, use incremental compilation on the modules with the new functionality to preserve the previous optimizations. The incremental compilation feature available in the Quartus II software provides a solution to this problem.

With Quartus II incremental compilation, you can preserve results and performance of unchanged logic in your design as you make ECOs elsewhere. The incremental compilation feature enables you to reduce design iteration time up to 70% and reach timing closure more efficiently. Incremental compilation facilitates block-based design, and allows you to preserve performance for unchanged blocks of the design. You can also target optimization techniques, such as physical synthesis, to specific design blocks while leaving other blocks untouched.

In a typical design flow, a hierarchical design is flattened into a single netlist before logic synthesis and fitting; therefore, the entire design is recompiled every time the design changes. However, the incremental compilation feature allows you to partition a design along any of its hierarchical boundaries. The Quartus II software separately synthesizes and fits each individual hierarchical design partition. The Quartus II software combines or merges the design partitions to form a netlist for subsequent stages of the Quartus II compilation flow.

Figure 5–1. Design Flow to Support ECO Changes



ECO Support at the Netlist Level

For some ECO changes, making changes at the netlist level can be faster than at the HDL level. This happens when you debug the design on silicon and need a very fast turnaround to generate a programming file for debugging the system. Figure 5–1 shows the recommended design flow to support ECO changes at the netlist level.

A typical application occurs when you uncover a problem on the board and isolate the problem to the appropriate nodes or I/O cells on the PLD. You must be able to correct the functionality of the incorrect logic cell or the properties of the I/O cell quickly and generate a new programming file. When correcting the functionality, you can verify the operation of the change without modifying the HDL, and perform a synthesis and place-and-route operation. This minimizes the disruption to the board verification procedure.

If this quick fix works, you do not need to change the HDL source code and rerun place-and-route. You have the option to perform one of the following options:

- Document the change that has been made
- Easily recreate the steps taken to produce the changes to the design
- Generate EDA simulation netlists for verification of the design
- Perform static timing analysis on the design

The Chip Planner in the Quartus II software provides these capabilities. The Chip Planner allows you to make functional changes to individual logic cells and to the I/O cell and phase-locked loop (PLL) parameters. These changes are stored in the Quartus II Change Manager log.

After the changes have been implemented in the design for the fix, you can regenerate a netlist for EDA gate-level simulation. You can run gate-level simulation to ensure that the modified design meets the functional specifications. You can also run static timing analyses to verify that your design still meets the timing you had specified. If you rerun timing analysis to sign-off the design, you can rerun static timing analysis on the netlist containing the ECO changes.

For more information, refer to the *Design Analysis & Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*.

Conclusion

Support for ECOs requires a combination of a modular design methodology and the appropriate software design tools.

The Quartus II software provides you with the software tools and the design methodology to perform successful ECOs at both the HDL and netlist level for programmable logic designs. This reduces the design cycle time and provides faster timing closure on designs that require last-minute changes.

Document Revision History

The following table shows the revision history for this chapter.

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

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Minor updates for the Quartus II software version 6.1.0 release, including • Chapter 5 was formerly Chapter 4 in version 6.1.0 • Globally changed “Chip Editor” to “Chip Planner” • Consolidated (previously) Figure 4–1 with the current Figure 5–1 (previously 4–2) to include information about ECO changes at both the HDL and Netlist levels	Beginning with the Quartus II software version 6.1, the Chip Editor tool has a new name, the Chip Planner. The Chip Planner allows you to perform ECOs at the netlist level.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0 release.	
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	
May 2005 v.5.0.0	Chapter 4 was formerly Chapter 3 in version 4.2.	

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

Date & Document Version	Changes Made	Summary of Changes
December 2004 v2.1	Updated for Quartus II software version 4.2: • Chapter 4 was formerly Chapter 3. • General formatting and editing updates. • Device family support descriptions updated. • Updated HardCopy structured support for performance improvements. • Quartus II Archive File automatically receives buffer insertion. • Power Calculator now Power Estimator for afflicted devices. • Updates to tables, figures • The description of How to Design HardCopy Stratix Devices was updated. • The description of HardCopy Timing Optimization Wizard was updated. • HardCopy Floorplans & Timing Modules was re-named to Design Optimization. • The description of Performance Estimation was updated. • Added new section of Buffer Insertion. • Location Constraints was updated. • Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices was removed. • A new section, Altera-Recommended HDL Coding Guidelines, was added. • Table 2-5 was added. It lists the HardCopy Stratix design files collected by the HardCopy Files Wizard. • The description of the HardCopy APEX Power Estimator was updated. • A new section on Targeting Designs to HardCopy APEX Devices was added.	
June 2004 v.2.0	New functionality for Quartus II software version 41. Updates to tables, figures.	
February 2004 v1.0	Initial Release.	

Section II. Design Guidelines

Today's programmable logic device (PLD) applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Designs coded optimally will behave in a predictable and reliable manner, even when re-targeted to different device families or speed grades. This section presents design and coding style recommendations for Altera® devices.

This section includes the following chapters:

- Chapter 6, Design Recommendations for Altera Devices
- Chapter 7, Recommended HDL Coding Styles

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

6. Design Recommendations for Altera Devices

QII51006-6.1.0

Introduction

Today's FPGA applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Well-coded designs behave in a predictable and reliable manner even when re-targeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and HardCopy® or ASIC implementations for prototyping and production.

For optimal performance, reliability, and faster time-to-market when designing with Altera® devices, you should:

- Understand the impact of synchronous design practices
- Follow recommended design techniques including hierarchical design partitioning
- Take advantage of the architectural features in the targeted device

This chapter presents design recommendations in these areas, and describes the Quartus II Design Assistant that can help you check your design for violations of design recommendations. The chapter contains the following sections:

For specific HDL coding examples and recommendations, including coding guidelines for targeting dedicated device hardware, such as memory and DSP blocks, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For information about migrating designs to HardCopy devices, refer to the *HardCopy Series Design Guidelines* chapter in the *HardCopy Series Handbook*.

Synchronous FPGA Design Practices

The first step in a good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you consistently meet your design goals. Problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades. In addition, if you plan to migrate your design to a high-volume solution such as Altera HardCopy devices, or if you are prototyping an ASIC, then synchronous design practices help ensure successful migration.

Fundamentals of Synchronous Design

In a synchronous design, everything is related to the clock signal. On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, the outputs of combinational logic feeding the data inputs of registers change values. This change triggers a period of instability due to propagation delays through the logic as the signals go through a number of transitions and finally settle to new values. Changes happening on data inputs of registers do not affect the values of their outputs until the next active clock edge.

Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input has been stable for at least the setup time of the register
- After an active clock edge, the data input remains stable for at least the hold time of the register

When you specify all your clock frequencies and other timing requirements, the Quartus® II Timing Analyzer issues actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin of your design. By meeting these external pin requirements and following synchronous design techniques, you ensure that you satisfy the setup and hold times for all registers within the Altera device.

To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feeds a register should have a synchronous relationship with the clock of the register. If signals are asynchronous, you can register the signals at the input of the Altera device to help prevent a violation of the required setup and hold times.

When the setup or hold time of a register is violated, the output can be set to an intermediate voltage level between the high and low levels, called a metastable state. In this unstable state, small perturbations like 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 time.

For details about timing requirements and analysis in the Quartus II software, refer to the *Classic Timing Analysis* or the *TimeQuest Timing Analysis* chapters in volume 3 of the *Quartus II Handbook*.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in programmable logic device (PLD) designs, enabling them to take "short cuts" to save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which can result in incomplete timing constraints and possible glitches and spikes. Because today's FPGAs provide many high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. Now it is more important to focus on design practices that help you meet design goals consistently than to save device resources using problematic asynchronous techniques.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. In these cases, race conditions can arise where the order of signal changes can affect the output of the logic. PLD designs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine the timing delay associated with a particular block of logic ahead of time. As devices become faster because of device process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Specific examples are provided in "Design Guidelines" on page 6–4. Relying on a particular delay also makes asynchronous designs very difficult to migrate to different architectures, devices, or speed grades.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, the timing-driven algorithms used by your synthesis and place-and-route tools may not be able to perform the best optimizations, and reported results may not be complete.

Some asynchronous design structures can generate harmful glitches, which are pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit a number of glitches before they settle to their new values. These glitches can propagate through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

Design Guidelines

When designing with HDL code, understanding how a synthesis tool interprets different HDL design techniques and what results to expect are important. Your design techniques can affect logic utilization and timing performance, as well as the design's reliability. This section discusses some basic design techniques that ensure optimal synthesis results for designs targeted to Altera devices while avoiding several common causes of unreliability and instability. Design your combinational logic carefully to avoid potential problems and pay attention to your clocking schemes so you can maintain synchronous functionality and avoid timing problems.

Combinational Logic Structures

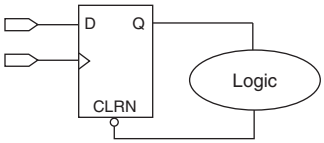
Combinational logic structures consist of logic functions that depend only on the current state of the inputs. In Altera FPGAs, these functions are implemented in the look-up tables (LUTs) of the device's architecture, using either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can also be used to implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs, and 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 6–1.

Figure 6–1. Combinational Loop through Asynchronous Control Pin



To perform timing analysis in the Quartus II software on asynchronous ports such as the `clear` or `reset`, on the Assignments menu, click **Settings**. In the Settings dialog box, select **Timing Requirements & Option** and click **More Settings**. Turn on **Enable Recovery/Removal Analysis**.

Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on the 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 combinational loop that holds the value of a signal until a new value is assigned. Latches can also be inferred from HDL code when you did not intend to use a latch. FPGA architectures are based on registers. In FPGA devices, latches actually use more logic resources and lead to lower performance than registers. This is different from other device architectures where latches may add less delay and can be implemented with less silicon area than registers.

Latches can cause various difficulties in the design. Although latches are memory elements, they are fundamentally different from registers. When a latch is in feed-through or transparent mode, there is a direct path

between the data input and the output. Glitches on the data input can pass through the output. The timing for latches is also inherently ambiguous. For example, when analyzing a design with a D-latch, the software cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design; therefore, another designer cannot easily modify the design or reuse the code.

In some cases, your synthesis tool can infer a latch that does not exhibit problems with glitches. Inferring the Altera `lpm_latch` function ensures that the implementation will be glitch-free in Altera architectures. Some third-party synthesis tools list the number of `lpm_latch` functions that are inferred. When using Quartus II integrated synthesis, these latches are reported in a section of the Compilation Report called **User-Specified and Inferred Latches**. If a latch or combinational loop in your design is not listed in this report, it means that it was not inferred as a "safe" latch by the software and is not considered glitch-free.

However, even glitch-free latches may not be analyzed completely during timing analysis. The Quartus II software provides an option called **Analyze latches as synchronous elements** that allows you to treat latches as start and end points for timing analysis (a typical analysis performed in FPGA design tools). With this option turned on, latches are analyzed as registers (with an inverted clock). The Quartus II software does not perform cycle-borrowing analysis, such as that performed by third-party timing analysis tools such as Synopsys PrimeTime.

In addition, latches have a limited support in formal verification tools. Therefore, it is especially important to ensure that you do not use latches when using formal verification.

Altera recommends avoiding using latches to ensure that you can completely analyze and verify the timing performance and reliability of your design.

Delay Chains

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

As described above, delays in PLD designs can change with each place-and-route cycle. Effects such as rise/fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. See "Hazards of

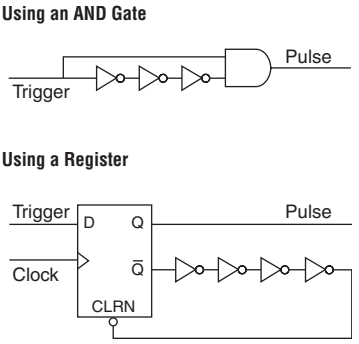
Asynchronous Design" on page 6–3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kind of problems.

In some ASIC designs, delays are used for buffering signals as they are routed around the device. This functionality is not needed in FPGA devices because the routing structure provides buffers throughout the device.

Pulse Generators & Multivibrators

Delay chains are sometimes used to generate either one pulse (pulse generators) or a series of pulses (multivibrators). There are two common methods for pulse generation, as shown in Figure 6–2. These techniques are purely asynchronous and should be avoided.

Figure 6–2. Asynchronous Pulse Generators



In "Using an AND Gate" (Figure 6–2), a trigger signal feeds both inputs of a 2-input `AND` gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of input values. This technique artificially increases the width of the glitch by using a delay chain.

In "Using a Register" (Figure 6–2), 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.

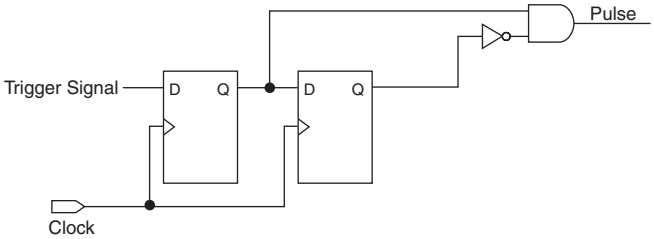
The width of pulses generated in this way are difficult for synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined after placement and routing, when routing

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, and 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 6–3.

Figure 6–3. 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 where 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. The following sections provide some specific examples and recommendations for avoiding these 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. See "Clock Network Resources" on page 6–40 for a detailed explanation.

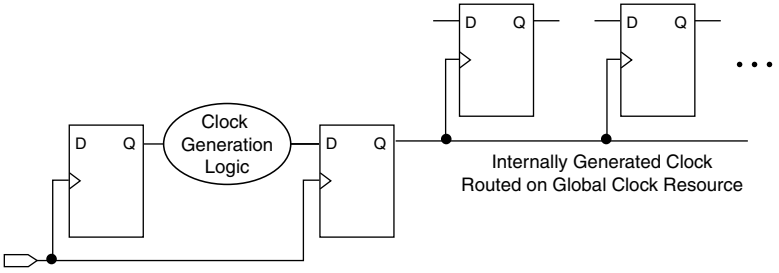
Avoid data transfers between different clocks wherever possible. If a data transfer between different clocks is needed, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a Clock Setup Uncertainty and Clock Hold Uncertainty value of 10% to 15% of the clock delay.

Internally Generated Clocks

If you use the output from combinational logic as a clock signal or as an asynchronous reset signal, you should expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or an asynchronous input) to a register can have significant consequences. Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Because of these problems, always register the output of combinational logic before you use it as a clock signal. See Figure 6–4.

Figure 6–4. Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that the glitches generated by the combinational logic are blocked at the data input of the register.

Divided Clocks

Designs often require clocks created by dividing a master clock. Most Altera FPGAs provide dedicated phase-locked loop (PLL) circuitry for clock division. Using dedicated PLL circuitry can help you to avoid many of the problems that can be introduced by asynchronous clock division logic.

When you must use logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design so that registers always directly generate divided clock signals, as described in "Internally Generated Clocks" on page 6–9, and route the clock on global clock resources. To avoid glitches, you should not decode the outputs of a counter or a state machine to generate clock signals.

Ripple Counters

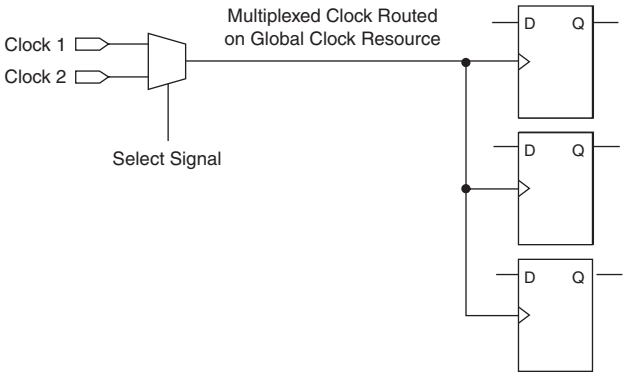
To simplify verification, Altera recommends avoiding ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks have to be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and place-and-route tools.

Ripple clock structures are often used to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus II software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows you to construct a counter using one logic element per counter bit. Altera recommends that you avoid using ripple counters under any circumstances.

Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source, as in Figure 6–5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 6–5. Multiplexing Logic & 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, 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, then 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 need 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 of the PLL in the

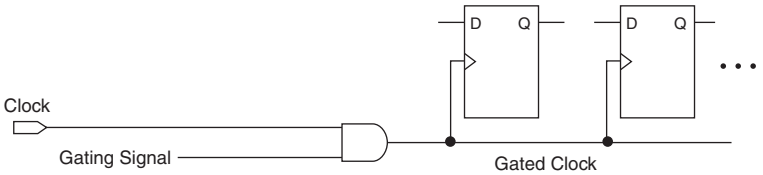
Stratix® series of devices, or the Clock Control Block in Stratix II, Stratix II GX, Hardcopy II, and Cyclone® II devices. These dedicated hardware blocks ensure that you use global low-skew routing lines and avoid any possible hold time problems on the device due to logic delay on the clock line.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as shown in Figure 6–6. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 6–6. 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 multiplexing logic, if it is available in your target device. For example, you can use the clock control block in Stratix II and Cyclone II devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew and avoid any possible hold time problems on the device due to logic delay on the clock line.

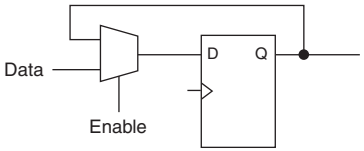
Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, you should use a synchronous scheme such as those described in the "Synchronous Clock Enables" section. For improved power reduction when gating clocks with logic, refer to "Recommended Clock-Gating Methods" on page 6–14.

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 will perform 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 6–7).

Figure 6–7. Synchronous Clock Enable

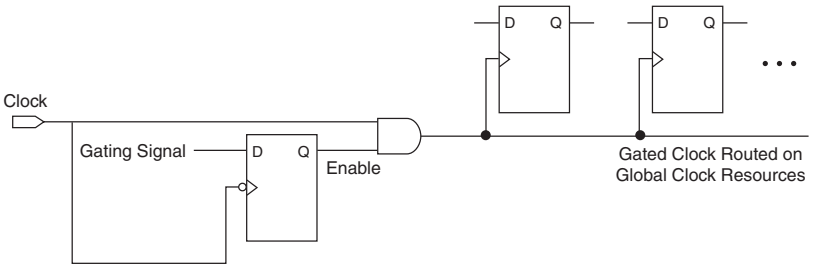


Recommended Clock-Gating Methods

Only use gated clocks 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 6–8 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 6–8. Recommended Clock Gating Technique



In the technique shown in Figure 6–8, 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 6–8). 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 signal must be generated in 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 6–8, apply a clock setting to the output of the AND gate. Otherwise, the Timing Analyzer may analyze the circuit using the clock path through the register as the longest clock path and the path that skips the register as the shortest clock path, resulting in artificial clock skew.

Checking Design Violations Using the Design Assistant

To improve the reliability, timing performance, and logic utilization of your design, practicing good design methodology and understanding how to avoid design rule violations are important. The Quartus II software provides a tool that automatically checks for design rule violations, and tells you where they occur.

The Design Assistant is a design-rule checking tool that allows you to check for any possible design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines or design rules. 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 want to allow these rule violations. The Design Assistant generates design violation reports with clear details about each violation, based on the settings you specified.

The first parts in this section provide an introduction to the Quartus II design flow with Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last few parts of the section describe the design rules and details about the reports generated by the Design Assistant.

Quartus II Design Flow with the Design Assistant

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

To set the Design Assistant to run automatically during compilation, on the Assignments menu, click **Settings**. In the Category list, click **Design Assistant**. Turn on **Run Design Assistant during compilation** (Figure 6–9). 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 options for the rules that you do not want the Design Assistant to use. Refer to "The Design Assistant Page" on page 6–18.

Figure 6–9. Set the Design Assistant to Run Automatically during Compilation

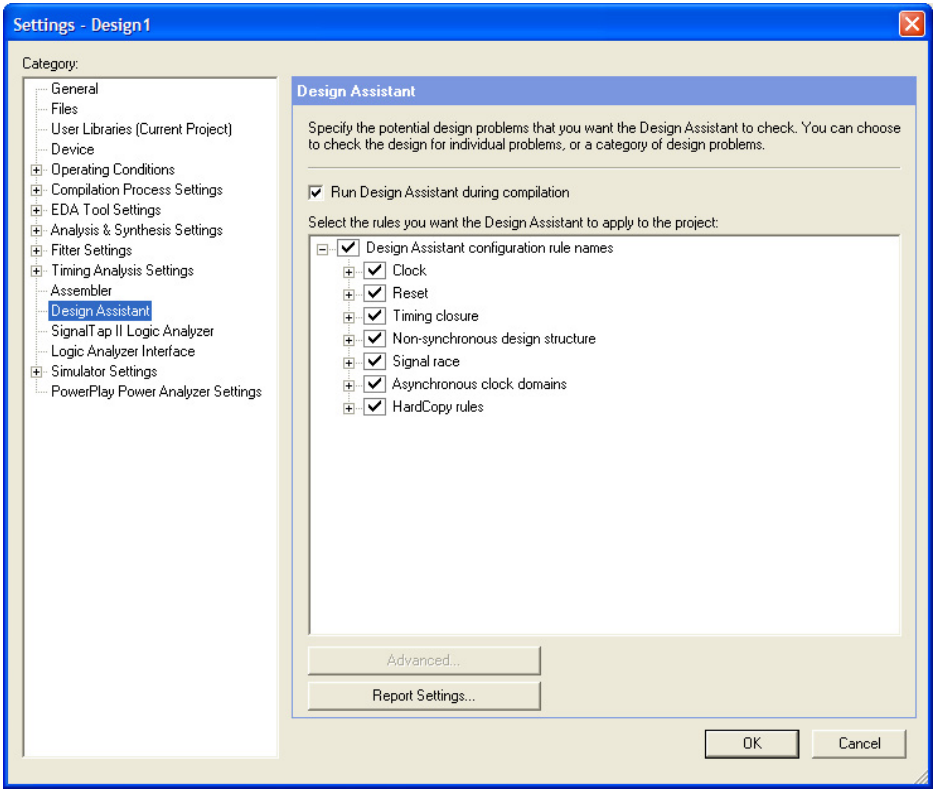
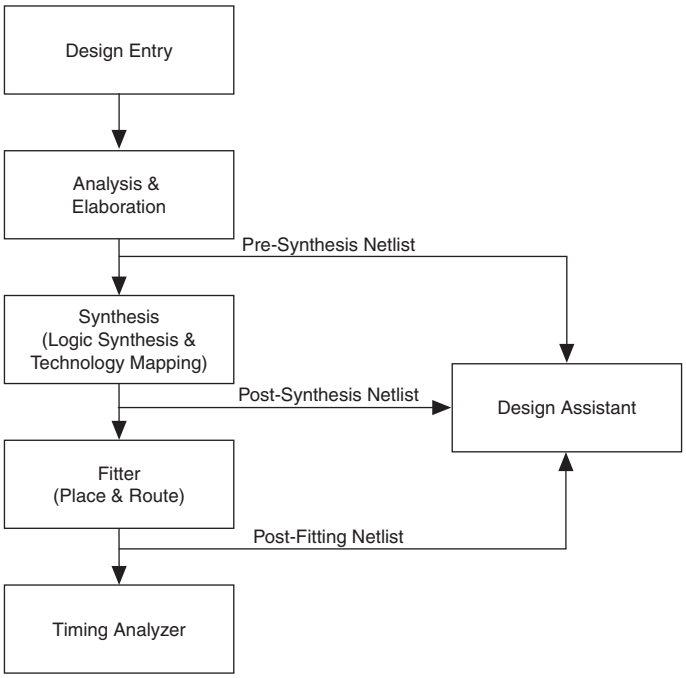


Figure 6–10 shows the Quartus II software design flow with Design Assistant.

Figure 6–10. Quartus II Design Flow with the Design Assistant



The Design Assistant analyzes your design netlist at different stages in 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 netlist may be different due to the optimization 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 netlist or a post-fitting netlist.

When you run the Design Assistant after running a full compilation or fitting, the Design Assistant performs a post-fitting analysis on the design. When you start the Design Assistant after performing Analysis & Synthesis, the Design Assistant performs post-synthesis analysis on the design. When you start the Design Assistant after performing Analysis & 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 `-rtl` option with the `quartus_drc` executable.

The Design Assistant generates warning messages when your design violates design rules, and generates information messages to provide information regarding the rules. The Design Assistant supports any Altera device supported by the Quartus II software.

The Design Assistant Page

To specify which rules you want the Design Assistant to apply to your design, on the Assignments menu, click **Settings**. In the Settings dialog box, in the Category list, select **Design Assistant**. In the Design Assistant page, turn on the rule which you want the Design Assistant to apply during analysis (Figure 6–9). By default, all of the rules are turned on.

In the Design Assistant page, in the Timing Closure category, if **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out** are turned on, you can use the **High Fan-Out Net Settings** dialog box to specify the number of fan-out a node must have to be reported by the Design Assistant. To open the **High Fan-Out Net Settings** dialog box, in the Design Assistant page, in the Timing Closure category, select **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out**. Click on the **High Fan-Out Net Settings** button.

In the Design Assistant page, in the Clock category, if you turn on **Clock signal should be a global signal**, you can use the **Global Clock Threshold Settings** dialog box to specify the number of nodes with the highest fan-out which you want the Design Assistant to report. To open the **Global Clock Threshold Settings** dialog box, on the Design Assistant page, in the Clock category, select **Clock signal should be a global signal**. Click on the **Global Clock Threshold Settings** button.

To specify the maximum number of messages to be reported by the Design Assistant, on the Design Assistant page, click on the **Report Settings** button, and fill in the maximum number of violation messages and detail messages to be reported.

Message Severity Levels

The Design Assistant classifies messages and rules using the four severity levels described in Table 6–1. Following Altera guidelines is very important for designs that will be migrated to the HardCopy series of devices, therefore the table highlights the impact of a rule violation on a HardCopy migration. Designs that adhere to Altera recommended design guidelines do not produce any messages with critical, high, or medium level of severity.

Table 6–1. Design Assistant Message Severity Levels

Severity Level	Explanation
Critical	A violation of the rule critically affects the reliability of the design. Altera may not be able to implement the design successfully without closely reviewing the violations with the designer for HardCopy devices conversions.
High	A violation of the rule affects the reliability of the design. Altera must review the violation before implementing the design for HardCopy device conversion.
Medium	The rule violation may result in implementation complexity which may have an impact for HardCopy device conversions.
Information Only	The rule provides information regarding the design.

Design Assistant Rules

This section describes the Design Assistant rules and details some of the reasons that Altera recommends following certain guidelines. Many of the Design Assistant rules enforce the design guidelines discussed in previous sections of this chapter.

Every rule is represented by a rule ID and has its own severity level. The rule ID is normally used in Tcl commands for rule suppression. The letter in each rule ID corresponds to the group of rules based on the following scheme.

- **A**—Asynchronous design structure rules
- **C**—Clock rules
- **R**—Reset rules
- **S**—Signal race rules
- **T**—Timing closure rules
- **D**—Asynchronous clock domain rules
- **H**—HardCopy rules

For example, the rule "The Design Should Not Contain Combinational Loops" is the first rule in the asynchronous design structure rules; therefore it is represented by rule ID A101.

Summary of Rules & IDs

Table 6–2 lists the rules, their rule IDs, and their severity level.

Table 6–2. Summary of Rules & IDs (Part 1 of 2)

Rule ID	Rule Name	Severity Level
A101	The Design Should Not Contain Combinational Loops	Critical
A102	The Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic	Critical
A103	The Design Should Not Contain Delay Chains	High
A104	The Design Should Not Contain Ripple Clock Structures	Medium
A105	Pulses Should Not Be Implemented Asynchronously	Critical
A106	Multiple Pulses Should Not Be Generated in the Design	Critical
A107	The Design Should Not Contain SR Latches	High
A108	The Design Should Not Contain Latches	High
A109	Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM	Medium
A110	The Design Should Not Contain Asynchronous Memory	Medium
C101	Gated Clocks Should Be Implemented According to the Altera Standard Scheme	Critical
C102	Logic Cells Should Not Be Used to Generate Inverted Clock	High
C103	The Input Clock Pin Should Fan Out to Only One Set of Clock Gating Logic	High
C104	Clock Signal Source Should Drive Only Input Clock Ports	Medium
C105	The Clock Signal Should Be a Global Signal	High
C106	Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges	Medium
R101	Combinational Logic Used as a Reset Signal Should Be Synchronized	High
R102	External Reset Should Be Synchronized Using Two Cascaded Registers	Medium
R103	External Reset Should Be Synchronized Correctly	High
R104	The Reset Signal Generated in One Clock Domain & Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly	High
R105	Asynchronous Clock Domains Should Be Synchronized If the Reset Signal Is Generated in One Clock Domain & Used in Another	Medium
S101	The Output Enable & Input of the Same Tri-state Nodes Should Not Be Driven by the Same Signal Source	High
S102	The Synchronous Port & Reset Port of the Same Register Should Not Be Driven by the Same Signal Source	High
T101	Nodes with More Than Specified Number of Fan-outs: <n>	Information Only

Table 6–2. Summary of Rules & IDs (Part 2 of 2)

Rule ID	Rule Name	Severity Level
T102	Top Nodes with Highest Fan-out: <n>	Information Only
D101	Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains	High
D102	Multiple Data Bits Transferred across Asynchronous Clock Domains Are Synchronized, But All Bits Are Not Aligned in the Receiving Clock Domain	Medium
D103	Data Bits Are Not Correctly Synchronized When Transferred between Asynchronous Clock Domains	High
H101	Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank	Medium
H102	A PLL Drives Multiple Clock Network Types	Medium

The Design Should Not Contain Combinational Loops

Severity Level: Critical
Rule ID: A101

A combinational loop is created by establishing a direct feedback loop on combinational logic that is not synchronized by a register. A combinational loop also occurs when the output of a register is fed back to an asynchronous pin of the same register through combinational logic. Combinational loops are among the most common causes of instability and reliability in your designs, and should be avoided whenever possible. Refer to "Combinational Loops" on page 6–4 for examples of the kinds of problems that combinational loops can cause.

The Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic

Severity Level: Critical
Rule ID: A102

A combinational loop occurs when you feed back the output of a register to an asynchronous pin of the same register (for example, the register's preset or asynchronous load signal), or the register drives combinational logic that drives one of the control signals on the same register. Combinational loops are among the most common causes of instability and reliability in your designs, and should be avoided whenever possible. Refer to "Combinational Loops" on page 6–4 for examples of the kinds of problems that combinational loops can cause.

The Design Should Not Contain Delay Chains

Severity Level: High
Rule ID: A103

Delay chains are created when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Delay chains are sometimes used to create intentional delay to resolve race conditions. Delay chains may cause significant problems as it affects the rise and fall time differences in your design.

This rule applies only for delay chains implemented in logic cells. Delay chains in I/O portions of the device are not detected by the Design Assistant. Logic cells created by compiler process (for example, synthesis or fitting) will not be reported the Design Assistant, either. Altera does not recommend instantiating a cell that does not benefit a design, and is used only to delay the signal. Refer to "Delay Chains" on page 6–6 for examples of the kinds of problems that delay chains can cause.

The Design Should Not Contain Ripple Clock Structures

Severity Level: Medium
Rule ID: A104

Designs should not contain ripple clock structures. These structures use two or more cascaded registers in which the output of each register feeds the clock pin of the register in the next stage. Cascading structures cause large skew in the output signal because each stage of the structure causes a new clock domain to be defined. The additional clock domains from each stage of the ripple clock are difficult for static timing analysis tools to analyze. Refer to "Ripple Counters" on page 6–10 for examples of the kinds of problems that ripple clock structures can cause.

Pulses Should Not Be Implemented Asynchronously

Severity Level: Critical
Rule ID: A105

There are two common methods for pulse generation:

- Increasing the width of a glitch using a 2-input AND, NAND, OR, or NOR gate, where the source for the two gate inputs are the same, but one of the gate inputs is inverted
- Using a register where the register output drives the register's own asynchronous reset signal through a delay chain (refer to "Delay Chains" on page 6–6 for more details).

These techniques are purely asynchronous and therefore should be avoided. Refer to "Pulse Generators & Multivibrators" on page 6–7 for recommended pulse generation guidelines.

Multiple Pulses Should Not Be Generated in the Design

Severity Level: Critical
Rule ID: A106

A common asynchronous multiple-pulse-generation technique consists of a combinational logic gate in which the inverted output feeds back to one of the inputs of the same gate. This feedback path causes a combinational loop which forces the output to change state, and therefore oscillate. Sometimes multiple pulse generators or multivibrator circuits are built out of a series of cascaded inverters in a structure called a "ring oscillator". Oscillation creates new artificial clock in your design that is difficult for the Quartus II software to determine, set, or verify.

Structures that generate multiple pulses cause more problems than pulse generators because of the number of pulses involved. In addition, multi-pulse generators also increase the frequency of the design. See "Pulse Generators & Multivibrators" on page 6–7 for recommended pulse generation guidelines.

The Design Should Not Contain SR Latches

Severity Level: High
Rule ID: A107

A latch is a combinational loop that holds the value of a signal until a new value is assigned. Combinational loops are hazardous to your design and are the most common causes of instability and reliability. Refer to "Combinational Loops" on page 6–4 for examples of the kinds of problems that combinational loops can cause.

Your design should not contain SR latches. An SR latch can cause glitches and ambiguous timing which complicates timing analysis of your design. Refer to "Latches" on page 6–5 for details about latches, and for more examples of the kinds of problems that latches can cause.

The Design Should Not Contain Latches

Severity Level: High
Rule ID: A108

The Design Assistant generates this rule when it identifies one or more structures as latches but cannot determine the latch types. As a result, this rule is applied when the Design Assistant identifies a 2-input latch structure that does not resemble an SR latch. The latches may also be part of more sophisticated latches that the Design Assistant cannot identify.

The Design Assistant generates this rule only when it identifies one or more structures in your design that are latches. Refer to "Latches" on page 6–5 for details about latches, and for examples of the kinds of problems that latches can cause.

The difference between A107 ("The Design Should Not Contain SR Latches") and A108 is that A107 triggers only when an SR latch is detected. A108 triggers when there's an unidentified latch in your design.

Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM

Severity Level: Medium
Rule ID: A109

Altera FPGA devices contain flexible embedded memory structures that can be configured into many different modes. One possible mode is asynchronous RAM. The definition of an asynchronous RAM circuit is one in which the write-enable signal driving into the RAM causes data to be written into it without a clock being required.

You should not use combinational logic to directly drive the write-enable signal of an asynchronous RAM. Any glitches that exist on the write-enable signal can cause the asynchronous RAM to be corrupted. Also, the data and write address ports of the RAM should be stable before the write pulse is asserted, and must remain stable until the write pulse is de-asserted. Because of the limitations to using memory structures in this asynchronous mode, synchronous memories are always preferred. In addition, synchronous memories provide higher design performance.

As a guideline, a register should be used between the combinational logic and the asynchronous RAM, or the asynchronous RAM should be replaced with synchronous memory. Refer to "Hazards of Asynchronous Design" on page 6–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

The Design Should Not Contain Asynchronous Memory

Severity Level: Medium
Rule ID: A110

You should avoid using asynchronous memory (for example, asynchronous RAM) in your design, because asynchronous memory can become corrupted by glitches created in the combinational logic that drives the write-enable signal of the memory. Asynchronous memory requires that the data and write address ports of the memory be stable before the write pulse is asserted, and must remain stable until the write pulse is de-asserted. In addition, asynchronous memory has lower performance than synchronous memory.

As a guideline, a register should be used between the combinational logic and the asynchronous RAM, or the asynchronous RAM should be replaced with synchronous memory. Immediately registering both input and output of the RAM improves performance and timing closure. Refer to "Hazards of Asynchronous Design" on page 6–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

Gated Clocks Should Be Implemented According to the Altera Standard Scheme

Severity Level: Critical
Rule ID: C101

Clock gating is sometimes used to turn parts of a circuit on and off to reduce the total power consumption of a device. Clock gating is implemented using an enable signal that controls some sort of gating circuitry. The gated clock signal prevents any of the logic driven by it from switching so the logic does not consume any power. For example, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive. However, the disadvantage of using this type of circuit is that it can lead to unexpected glitches on the resultant gated clock signal if certain rules are not followed.

Refer to "Gated Clocks" on page 6–12 for examples of the kinds of problems gated clock can cause. Refer to "Recommended Clock-Gating Methods" on page 6–14 for a recommended clock gating technique.

Logic Cells Should Not Be Used to Generate Inverted Clock

Severity Level: High
Rule ID: C102

Your design may require both positive and negative edges of a clock to operate. However, you should not implement an inverter to drive the clock input of a register in your design with a logic cell. Implementing the inverter with a logic cell can lead to clock insertion delay and skew, which is hazardous to your design and can cause problems with the timing closure of the design.

In addition, using a logic cell to implement an inverter is unnecessary. You should use the programmable clock inversion featured in the register to generate the inverted clock signal. Refer to "Clocking Schemes" on page 6–8 for details about different types of clocking methods.

The Input Clock Pin Should Fan Out to Only One Set of Clock Gating Logic

Severity Level: High
Rule ID: C103

Your design should not contain an input clock pin that fans out to more than one set of combinational logic used to drive a clock signal. To effectively reduce power consumption when combinational logic is used to drive a clock signal, Altera recommends that you ensure that all of the input clock pins in a design fan out to only one set of combinational logic. Refer to "Clocking Schemes" on page 6–8, and "Recommended Clock-Gating Methods" on page 6–14 for proper clock-gating techniques.

Clock Signal Source Should Drive Only Input Clock Ports

Severity Level: Medium
Rule ID: C104

Clock signal sources in a design should drive only input clock ports of registers. When a design contains clock signal sources that connect to ports other than the clock ports, the design is considered asynchronous and should be avoided. Asynchronous design structures can be hazardous to your design because some of them rely on the relative propagation delays of signals to function correctly, which can result in incomplete timing constraints and possible glitches and spikes. Refer to "Hazards of Asynchronous Design" on page 6–3 for examples of the kinds of problems that asynchronous design structures can cause. Also refer to "Clocking Schemes" on page 6–8 for proper clocking techniques.

This rule does not apply in the following conditions:

- When the clock signal source drives combinational logic that is used as a clock signal, and the combinational logic is implemented according to the Altera standard scheme
- When the clock signal source drives only a clock multiplexer that selects one clock source from a number of different clock sources

This type of multiplexer adds complexity to the timing analysis of a design. You should avoid using the multiplexer in the design.

- Using a clock multiplexer causes the "Gated Clocks Should Be Implemented According to the Altera Standard Scheme" rule (C101) to be enacted; refer to "Multiplexed Clocks" on page 6–11 for recommended clock multiplexing techniques

The Clock Signal Should Be a Global Signal

Severity Level: High
Rule ID: C105

You should ensure that all clock signals in your design use the global clock networks that exist in the target FPGA. Mapping clock signals to use non-dedicated clock networks can negatively affect the performance of your design. A non-global signal can be slower and have larger skew than a global signal because the clock must be distributed using regular FPGA routing resources.

To specify the number of minimum fan-outs that you want the Design Assistant to report, on the Design Assistant page, in the Clock category, select **Clock signal should be a global signal**. Click on the **Global Clock Threshold Settings** button, and enter the number in the dialog box.

If a design contains more clock signals than are available in the target device, you should consider reducing the number of clock signals in the design, such that only dedicated clock resources are used for clock distribution. However, if the design must use more clock signals than you can specify as global signals, implement the clock signals with the lowest fan-out using regular routing resources. Also, implement the fastest clock signals as global signals. Refer to "Clock Network Resources" on page 6–40 for detailed explanation about clock resources.

Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges

Severity Level: Medium
Rule ID: C106

This rule triggers an error message if your design contains a clock signal source that drives the clock inputs of both positive and negative edge-sensitive registers. This error also triggers if your design contains an inverted clock signal that drives the clock inputs of either positive or negative edge-sensitive registers.

These two scenarios can cause an increase in timing requirement complexity and difficulties in design optimization. Also, because those registers are clocked on the different edges, synchronous resetting is impossible. Refer to "Clocking Schemes" on page 6–8 for some specific examples and recommended clocking methods.

Combinational Logic Used as a Reset Signal Should Be Synchronized

Severity Level: High
Rule ID: R101

All combinational logic used to drive reset signals in your design should be synchronized. This means that a register should be placed between the combinational logic that drives reset signal and the input reset pin. Unsynchronized combinational logic can cause glitches and spikes that lead to unintentional reset signals. Synchronizing the combinational logic that drives the reset signal delays the resulting reset signal by an extra clock cycle and avoids unintentional reset. You should consider the extra clock cycle delay when using this method in your design.

External Reset Should Be Synchronized Using Two Cascaded Registers

Severity Level: Medium
Rule ID: R102

The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, the asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using a double-buffer circuit, which consists of two cascaded registers triggered on the same clock edge. Two cascaded registers are required to decrease

the probability of metastability in the reset domain. Both cascaded registers must be triggered in the same clock edge so the second register has enough time to resolve the metastable output from the first register.

External Reset Should Be Synchronized Correctly

Severity Level: High
Rule ID: R103

The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, the asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using two cascaded registers, and the registers should be triggered on the same clock edge.

This rule applies when an asynchronous reset or external reset signal is synchronized but fails to follow the recommended guidelines as described in rule R102 ("External Reset Should Be Synchronized Using Two Cascaded Registers"). This violation happens when the external reset is synchronized with only one register, or the cascaded synchronization registers are triggered on different clock edges.

R102 triggers when you don't use two cascaded registers to synchronize the external reset. R103 triggers when the external reset is synchronized but fails to follow the recommended guidelines.

The Reset Signal Generated in One Clock Domain & Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly

Severity Level: High
Rule ID: R104

If your design uses an internally generated reset signal generated in one clock domain and used in one or more other asynchronous clock domain, then the reset signal should be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, use the following guidelines:

- The reset signal should be synchronized with two or more cascading registers in the receiving asynchronous clock domain.
- The cascading registers should be triggered on the same clock edge.

- There should be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. The synchronization registers may sample unintended data due to the glitches caused by the logic.

This rule applies when the internal reset signal is synchronized but fails to follow the recommended guidelines. This happens when the external reset is only synchronized with one register, or the cascaded synchronization registers are triggered on different clock edges, or there is logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. Synchronizing the reset signal delays the signal by an extra clock cycle. You should consider this delay when using the reset signal in a design.

Asynchronous Clock Domains Should Be Synchronized If the Reset Signal Is Generated in One Clock Domain & Used in Another

Severity Level: Medium
Rule ID: R105

If your design uses an internally generated reset signal that is generated in one clock domain and used in one or more other asynchronous clock domain, then the reset signal should be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, you should follow guidelines described in Rule R104 ("The Reset Signal Generated in One Clock Domain & Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly").

This rule applies when the internally generated reset signal is not being synchronized.

The Output Enable & Input of the Same Tri-state Nodes Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S101

This rule applies when your design contains a tri-state node in which the input and output enable are driven by the same signal source. Signal race occurs between the input and output enable signals of the tri-state when they are propagated simultaneously. Race conditions lead to incorrect design function and unpredictable results. To avoid violation of this rule, the input and output enable of the tri-state should be driven by separate signal sources.

The Synchronous Port & Reset Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S102

A purely synchronous design is free of signal race conditions as long as the clock signal is properly distributed and the timing requirements of the registers are met. However, race conditions can typically occur when the synchronous input pin and the reset pin of the register are driven by the same signal source. Race conditions can cause incorrect design function and unpredictable results. As a guideline, the synchronous input pin and the reset pin of the register should be driven by separate signal sources.

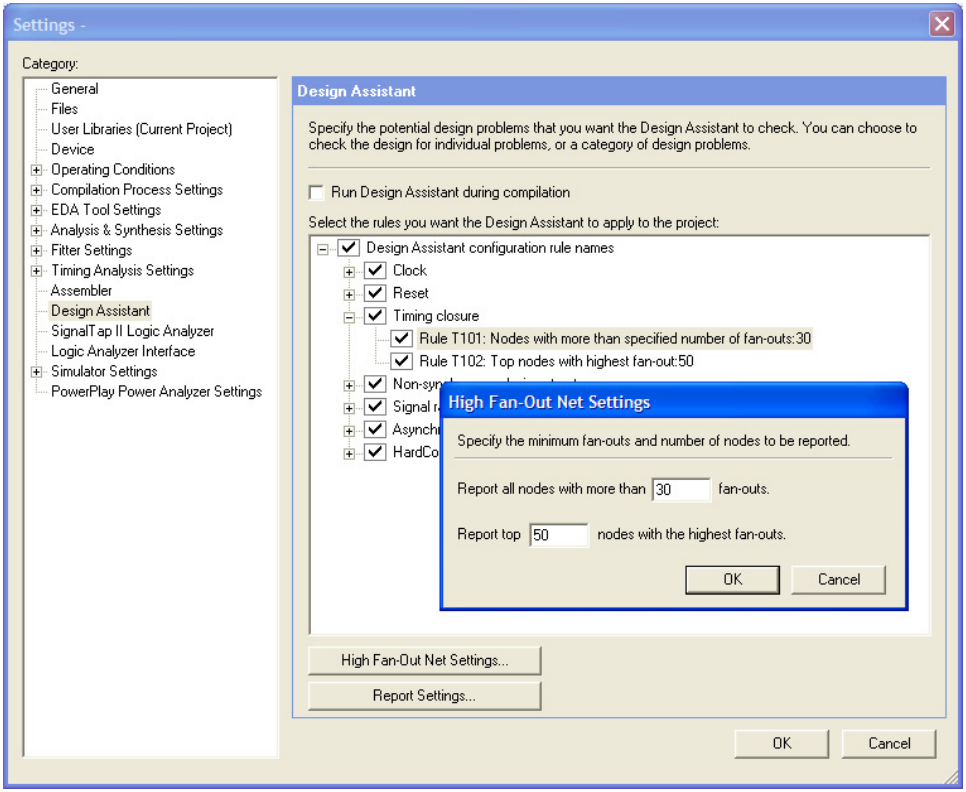
Nodes with More Than Specified Number of Fan-outs: <n>

Severity Level: Information Only
Rule ID: T101

This rule reports nodes that have more than a specified number of fan-outs, which can create timing challenges for your design.

To specify the number of fan-outs, on the Assignments menu, select **Settings**. In the Category list, select **Design Assistant**. On the Design Assistant page, expand the Timing Closure category by clicking the icon next to **Timing closure**. Turn on **Nodes with more than specified number of fan-outs**. Click the **High Fan-Out Net Settings** button. In the High Fan-Out Net Settings dialog box, enter the number of fan-outs a node must have to be reported by the Design Assistant (Figure 6–11).

Figure 6–11. Specifying the Number of Fan-Outs Reported by the Design Assistant



Top Nodes with Highest Fan-out: <n>

Severity Level: Information Only
Rule ID: T102

This rule reports the specified number of nodes with the highest fan-out, which can create timing challenges for your design.

To specify the number of fan-outs, on the Assignments menu, select **Settings**. In the Category list, select **Design Assistant**. On the Design Assistant page, click the ⊞ icon next to **Timing closure** to expand the folder. Select **Nodes with more than specified number of fan-outs**. Click the **High Fan-out Net Settings** button. In the High Fan-Out Net Settings dialog box, enter the number of nodes with the highest fan-out to be reported by the Design Assistant (Figure 6–11).

Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains

Severity Level: High
Rule ID: D101

The data bits transferred between asynchronous clock domains in a design should be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, then each data bit should be synchronized with two cascading registers in the receiving asynchronous clock domain, in which the cascaded registers are triggered on the same clock edge. There should be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

But if the data bits belong to multiple-bit data, a handshake protocol should be used to guarantee that all bits of the data bus are stable when the receiving clock domain samples the data. If a handshake protocol is used, only the data bits that act as `REQ` (request) and `ACK` (acknowledge) signals should be synchronized. The data bits that belong to multiple-bit data do not need to be synchronized. You can ignore the violation on the data bits that use a handshake protocol.

Multiple Data Bits Transferred across Asynchronous Clock Domains Are Synchronized, But All Bits Are Not Aligned in the Receiving Clock Domain

Severity Level: Medium
Rule ID: D102

This rule applies when all of the data bits that belong to multiple-bit data and are transferred between asynchronous clock domains are synchronized. However, not all data bits may be aligned in the receiving clock domain. Propagation delays may cause skew when the data reaches the receiving clock domain.

If the data bits belong to multiple-bit data and a handshake protocol is used, only the data bits that act as `REQ`, `ACK`, or both signals for the transfer should be synchronized with two or more cascading registers in the receiving asynchronous clock domain.

If all of the data bits belong to single-bit data, the synchronization of the data bits does not cause problems in the design.

Data Bits Are Not Correctly Synchronized When Transferred between Asynchronous Clock Domains

Severity Level: High
Rule ID: D103

The data bits that are transferred between asynchronous clock domains in a design should be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, then each data bit should be synchronized with two cascading registers in the receiving asynchronous clock domain. In this case, the cascaded registers are triggered on the same clock edge, and there should be no logic between the output of the transmitting clock domain. The cascaded registers in the receiving asynchronous clock domain.

This rule only applies when the data bits across asynchronous clock domains are synchronized but fail to follow the guidelines.

Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank

Severity Level: Medium
Rule ID: H101

If your design targets a HardCopy APEX™ 20K device, then you should not assign more than one `VREF` pin to a HardCopy test pin in an I/O bank in that targeted device. The assignment of more than one `VREF` pin to a HardCopy test pin can cause contention of the `VREF` bus.

You can find the list of HardCopy test pins in each of a HardCopy APEX 20K device's I/O banks in the Messages window, the Design Assistant Messages report, and the Design Assistant HardCopy Test Pins report. You should use this information to ensure that only one `VREF` pin is assigned to HardCopy test pins.

However, the Fitter may have assigned the `VREF` pins to the HardCopy test pins during compilation. To prevent the Fitter from making these assignments during the next compilation, create and assign the `VREF` pins manually instead of allowing the Fitter to do so automatically.

This rule only applies to designs that target HardCopy APEX 20K devices.

A PLL Drives Multiple Clock Network Types

Severity Level: Medium
Rule ID: H102

A PLL can compensate only one of the clock network types; therefore, the other non-compensated clock network types have a non-zero delay. However, the non-zero delay for the non- compensated clock network types can change between a Stratix device and its corresponding HardCopy Stratix device, or a Stratix II device and its corresponding HardCopy II device.

Therefore, if a Stratix FPGA design relies on the relative offset between the compensated clock network type and the non-compensated clock network types driven by a PLL, an error can occur in the corresponding HardCopy Stratix design because the relative offset in the HardCopy Stratix design may differ from the relative offset in the original Stratix FPGA design.

This rule only reports nodes in a design where a PLL drives multiple clock network types.

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 titled <*project_name*>**.drc.rpt** in the <*project_name*> subdirectory of the project directory.

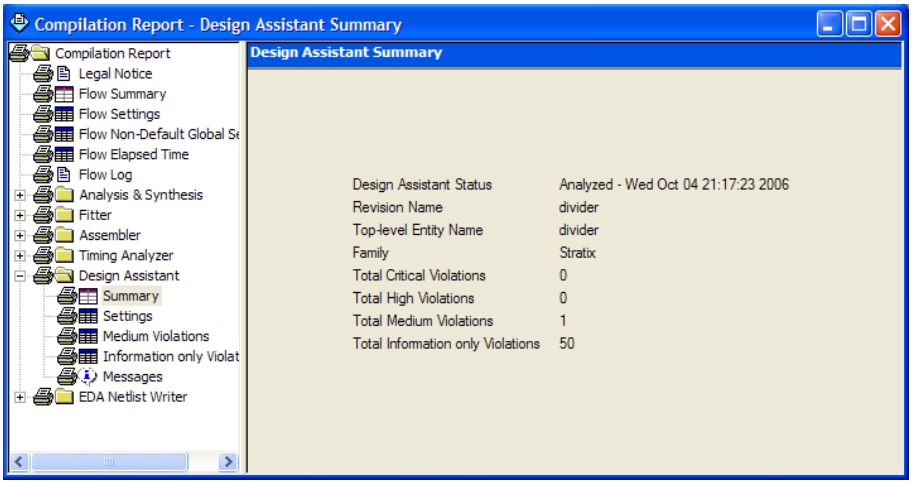
The Design Assistant generates the following five reports based on the settings specified in the Design Assistant page:

- Summary Report
- Settings Report
- Detailed Results Report
- Messages Report
- HardCopy Test Pins Report

Summary Report

The Design Assistant Summary report contains summary of the Design Assistant process on a particular project. This includes Design Assistant Status, Revision Name, Top-level Entity, Targeted Family Device, and total number of design violations of the project. Figure 6–12 shows the Design Assistant Summary report.

Figure 6–12. The Design Assistant Summary Report



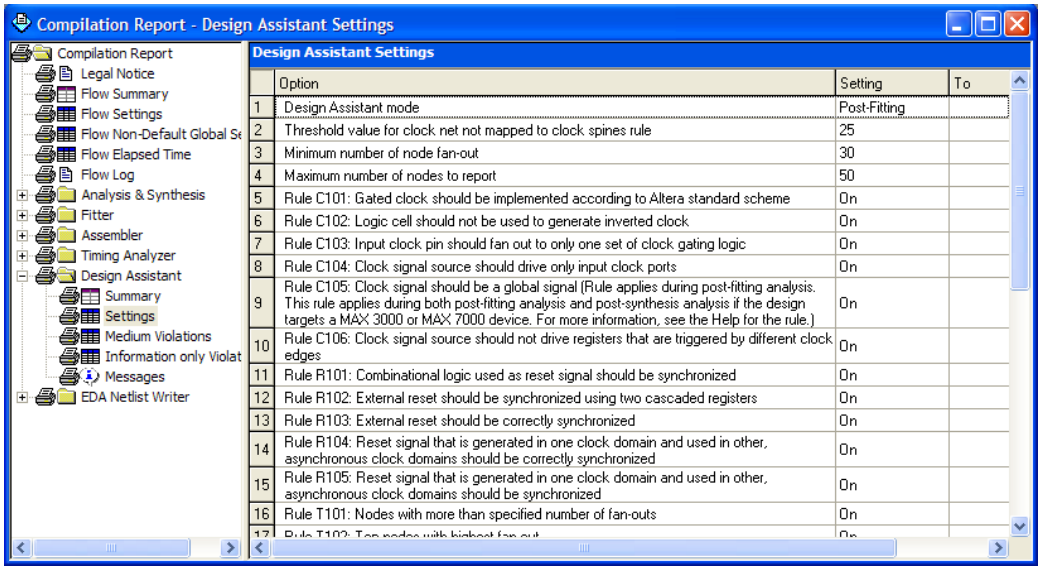
- **Design Assistant Status**—the status, end date, and end time of the Design Assistant operation.
- **Revision Name**—the revision name specified in the Revisions dialog box.
- **Top-level Entity Name**—the top-level entity of your design
- **Family**—the device family name specified in the Device page of the Settings dialog box.
- **Total Critical Violations**, **Total High Violations**, **Total Medium Violations**, and **Total Information Only Violations**—the total violations of the rules organized by level, some of which might affect the reliability of the design

Note that designs converted for HardCopy devices may not convert successfully without first closely reviewing the violations.

Settings Report

The Design Assistant Settings report contains a list of enabled Design Assistant rules and options that you specified in the Design Assistant page, as shown in Figure 6–13.

Figure 6–13. The Design Assistant Settings Report

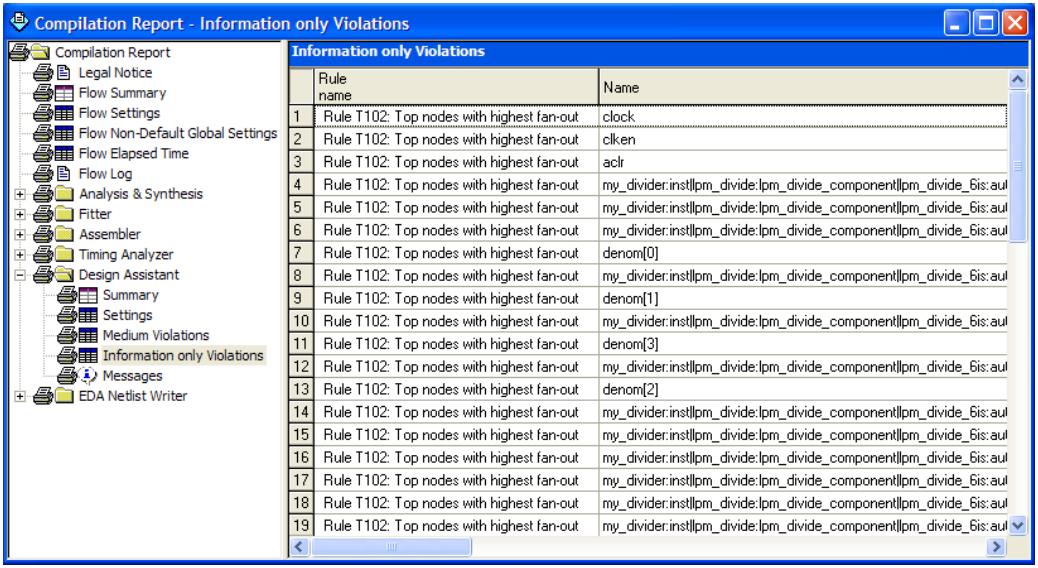


Detailed Results Report

The Detailed Results report contains detailed information of every rule violation including the rule name, the node name, and the fan-out. This report only appears if you specify settings in the Design Assistant page. Refer to "The Design Assistant Page" on page 6–18 for more information about how to specify the settings.

Separate Detailed Results reports are generated for critical, high, medium, and information only results. Figure 6–14 shows the Information Only Violations report.

Figure 6–14. The Design Detailed Results Report, Information Only



Messages Report

The Messages report contains current information, warning, and error messages generated during the Design Assistant process. You can right-click a message in the Messages report and click **Help** to display the Quartus II software Help with details about the selected message, or click **Locate** to trace or cross-probe the selected node and locate the source of the violation.

HardCopy Test Pins Report

The HardCopy Test Pins report appears only if you turn on the **Run Design Assistant during compilation** in the Design Assistant page, and if your design violates the "Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank" rule. The report lists all the HardCopy design rule violations, and also list all of the test pins in the HardCopy device.

Hierarchical Design Partitioning

A hierarchical design consists of multiple design blocks linked together in a hierarchy. When a design is partitioned hierarchically, you can compile, optimize and simulate the individual design blocks separately. You can use incremental compilation to follow a block-based design methodology where each block is placed and routed independently, then all blocks in the hierarchy are combined at the top level. Some synthesis tools have features to help you create separate netlist files or maintain separate parts of a netlist file for different parts of your design, to support block-based design techniques or incremental compilation.

For information about incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about incremental synthesis flows in your synthesis tool, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II handbook*.

When using a hierarchical or incremental design methodology, consider how the design is partitioned to achieve good results.

Altera recommends the following practices for partitioning designs:

- Partition the design at functional boundaries.
- Minimize the I/O connections between different partitions.
- Register all inputs and outputs of each block. This makes logic synchronous and avoids glitches and avoids any delay penalty on signals that cross between partitions. Registering I/Os typically eliminates the need to specify timing requirements for signals that connect between different blocks.
- Do not use "glue logic" or connection logic between hierarchical blocks. When you preserve hierarchy boundaries, glue logic is not merged with hierarchical blocks. Your synthesis software may optimize glue logic separately, which can degrade synthesis results and is not efficient when used with the LogicLock design methodology.
- Remember that logic is not synthesized or optimized across partition boundaries, which means any constant values (signals set to GND, for example) will not be propagated across partitions.
- Do not use tri-state signals or bidirectional ports on hierarchical boundaries. If you use boundary tri-states in a lower-level block, synthesis pushes the tri-states through the hierarchy to the top-level to take advantage of the tri-state drivers on the output pins of Altera device. Because this requires optimizing through hierarchies, lower-level boundary tri-state signals have restrictions with block-level design methodologies.
- Limit clocks to one per block. Partitioning the design into clock domains makes synthesis and timing analysis easier.

- Place state machines in separate blocks to speed optimization and provide greater encoding control.
- Separate timing-critical functions from non-timing-critical functions.
- Limit the critical timing path to one hierarchical block. You can group the logic from several design blocks to ensure the critical path resides in one block.

For more guidelines for creating design partitions for Quartus II incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Targeting Clock & Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the device architecture in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Clock Network Resources

Altera FPGAs provide device-wide global clock routing resources and dedicated inputs. You should use the FPGA's low-skew, high fan-out, dedicated routing where available. By assigning a clock input to one of these dedicated clock pins or using a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

In ASIC design, balancing the clock delay as it is distributed across the device can be important. Because Altera FPGAs provides device-wide global clock routing resources and dedicated inputs, there is no need to manually balance delays on the clock network.

Altera recommends limiting the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems. In addition, when you use combinational logic to generate an internal clock, it adds delays on the clock line. In some cases, delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register (such as hold time requirements) are violated and the design will not function correctly.

Today's FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are typically organized into a

hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically a number of dedicated clock pins to drive either the global or regional clock networks and both PLL outputs and internal clocks can drive various clock networks.

To reduce the clock skew within a given clock domain and ensure that hold times are met within that clock domain, assign each clock signal to one of the global high fan-out, low-skew clock networks in the FPGA device. Quartus II 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. To make explicit **Global Signal** logic option settings, on the Assignment 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) should 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 the 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 it can complicate timing analysis. It is not a recommended practice.

Reset Resources

ASIC designs may use local resets to avoid long routing delays on the signal. You should 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.

Register Control Signals

Avoid using an asynchronous load signal if the design's target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of those control signals. APEX™ 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 place-and-route software must use combinational logic to implement the same functionality. In addition, if you use signals in a priority other than the inherent priority in the device architecture, combinational logic may be required to implement the desired control signals. The 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*.

Conclusion

Following the design practices outlined in this chapter can help you meet your design goals consistently. Asynchronous design techniques may result in incomplete timing analysis, may clause 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 your quality of results.

Document Revision History

The following table shows the revision history for this chapter.

Table 6–3. Documentation Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes

November 2006
v6.1.0
	Added the following sections (with additional subsections):
	• "Checking Design Violations Using the Design Assistant"
	  • "Quartus II Design Flow with the Design Assistant"
	  • "The Design Assistant Page"
	  • "Message Severity Levels"
	  • "Design Assistant Rules"
	  • "Viewing Design Assistant Results"
		Quartus II software version 6.1 added the Design Assistant; the bulk of the changes to this chapter are related to this update.

May 2006 v6.0.0	Minor updates for the Quartus II version 6.0.	
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	
May 2005 v.5.0.0	Chapter 5 was formerly Chapter 4 in version 4.2.	
December 2004 v2.1	Updated for Quartus II software version 4.2: • Chapter 5 was formerly Chapter 6 in version 4.1. • General formatting and editing updates. • Updated hardware requirements for the Quartus II Timing Analyzer. • Added timing requirements and analysis details. • Updated Design Guidelines. • Added information about performing timing analysis on asynchronous ports. • Added inferred latches information. • Updated Delay Chains description. • Updated figures, tables. • Added Clocking Schemes information. • Added details to Multiplexed Clocks details. • Added clock gating details. • Updated Hierarchical Design Partitioning to include synthesis and incremental synthesis. • Added global routing information.	

Table 6–3. Documentation Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
June 2004 v.2.0	• Updates to tables, figures, coding examples. • New functionality for Quartus II software 4.1.	
February 2004 v1.0	Initial release.	

7. Recommended HDL Coding Styles

QII51007-6.1.0

Introduction

HDL coding styles can have a significant effect on the quality of results that you achieve for programmable logic designs. Synthesis tools optimize HDL code for both logic utilization and performance. However, sometimes the best optimizations require human understanding of the design, and synthesis tools have no information about the purpose or intent of the design. You are often in the best position to improve your quality of results.

This chapter addresses HDL coding style recommendations to ensure optimal synthesis results when targeting Altera® devices, including the following sections:

For additional guidelines on structuring your design, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

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*.

Using Altera Megafunctions

Altera provides parameterizable megafunctions that are optimized for Altera device architectures. Using megafunctions instead of coding your own logic saves valuable design time. Additionally, the Altera-provided megafunctions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size and set various options by setting parameters. Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions.

To use megafunctions in your HDL code, you can instantiate them as described in "Instantiating Altera Megafunctions in HDL Code". Sometimes it is preferable to make your code independent of device family or vendor, and you do not want to instantiate megafunctions directly. In cases where you do not want to instantiate a megafunction, follow the guidelines and coding examples in "Inferring Altera Megafunctions from HDL Code" on page 7–6 to ensure your generic HDL code infers the appropriate Altera megafunction.

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

For some designs, generic HDL code can provide better results than instantiating a megafunction. Refer to the following general guidelines and examples that describe when to use standard HDL code and when to use megafunctions:

- For simple addition or subtraction functions, use the + or − symbol instead of an LPM function. Instantiating an LPM function for simple arithmetic operations can result in a less efficient result because the function is hard coded and the synthesis algorithms cannot take advantage of basic logic optimizations.
- For simple multiplexers and decoders, use array notation (such as `out = data[sel]`) instead of an LPM function. Array notation works very well and has simple syntax. You can use the `lpm_mux` function to take advantage of architectural features such as cascade chains in APEX™ series devices, but use the LPM function only if you understand the device architecture in detail and want to force a specific implementation.
- Avoid division operations where possible. Division is an inherently slow operation. Many designers use multiplication creatively to produce division results.

Instantiating Altera Megafunctions in HDL Code

The following sections describe how to use megafunctions by instantiating them in your HDL code with the following methods:

- "Instantiating Megafunctions Using the MegaWizard Plug-In Manager"—You can use the MegaWizard® Plug-In Manager to parameterize the function and create a wrapper file.
- "Creating a Clear Box Netlist File for Other Synthesis Tools"—You can optionally create a clear box body instead of a wrapper file.
- "Instantiating Megafunctions Using the Port & Parameter Definition"—You can instantiate the function directly in your HDL code.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Use the MegaWizard Plug-In Manager as described in this section to create megafunctions in the Quartus II GUI that you can instantiate in your HDL code. The MegaWizard Plug-In Manager provides a graphical user interface 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 to be generated. Depending on which language you choose, the MegaWizard Plug-In Manager instantiates the megafunction with the correct parameters and generates one of the following sets of files:

- AHDL Text Design File (**.tdf**) wrapper file and a sample instantiation template Text Design File (**_inst.tdf**).
- Verilog HDL (**.v**) wrapper file, a sample instantiation template Verilog HDL file (**_inst.v**), and a black-box Verilog HDL module declaration.
- VHDL (**.vhd**) wrapper file and a sample instantiation template VHDL file (**_inst.vhd**).

You can instantiate the megafunction wrapper file in your design using the corresponding sample instantiation file. In addition, the MegaWizard Plug-In Manager can create the following files, if requested:

- 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**) and as a reference for Verilog HDL design files

Refer to Table 7–1 for a list and description of files generated by the MegaWizard Plug-In Manager.

Table 7–1. MegaWizard Plug-In Manager Generated Files

File	Description
<*output file*>**.bsf**	Block Symbol File—Used in the Quartus II Block Design Files (**.bdf**).
<*output file*>**.cmp**	Component Declaration File—Used in VHDL designs.
<*output file*>**.inc**	ADHL Include File—Used in AHDL designs.
<*output file*>**.tdf** *(1)*	AHDL Wrapper File—Megafunction wrapper file for instantiation in an AHDL design.
<*output file*>**.vhd** *(2) (4)*	VHDL Wrapper File—Megafunction wrapper file, or clear box netlist file, for instantiation in a VHDL design.
<*output file*>**.v** *(3) (4)*	Verilog HDL Wrapper File—Megafunction wrapper file, or clear box netlist file, for instantiation in a Verilog HDL design.
<*output file*>**_bb.v** *(3)*	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*>**_inst.tdf** *(1)*	Text Design File Instantiation Template—Sample AHDL instantiation of the subdesign in the megafunction wrapper file.
<*output file*>**_inst.vhd** *(2)*	VHDL Instantiation Template—Sample VHDL instantiation of the entity in the megafunction wrapper file.
<*output file*>**_inst.v** *(3)*	Verilog HDL Instantiation Template—Sample Verilog HDL instantiation of the module in the megafunction wrapper file.

Notes to Table 7–1:

(1) The MegaWizard Plug-In Manager generates this file only if you select AHDL output files.

(2) The MegaWizard Plug-In Manager generates this file only if you select VHDL output files.

(3) The MegaWizard Plug-In Manager generates this file only if you select Verilog HDL output files.

(4) A megafunction wrapper file is created by default for most megafunctions. To take advantage of the clear box feature, on the Tools menu, click **MegaWizard Plug-In Manager** and turn on **Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)**. For additional information about how to use the MegaWizard Plug-In Manager, refer to the Quartus II Help.

Creating a Clear Box Netlist File for Other Synthesis Tools

When you use certain megafunctions with other synthesis tools, you can optionally create a clear box body instead of a wrapper file. The clear box body is a fully synthesized megafunction that you can use with certain other EDA synthesis tools.

The netlist file that contains the megafunction clear box body provides your third-party synthesis tool with information about the architectural details used in the Quartus II software. This information enables certain synthesis tools to better report timing and resource utilization estimates. In addition, synthesis tools can use the timing information to focus timing-driven optimizations and improve the quality of results.

For information about clear box support 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*.

To generate a clear box netlist, turn on **Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** on the megafunction selection page 2a of the MegaWizard Plug-In Manager.

Note that not all megafunctions support clear box netlists. If you cannot create a clear box netlist for a particular megafunction, the option to generate the netlist is not shown on page 2a of the MegaWizard Plug-In Manager. Some megafunctions always use a clear box netlist file, in which case the option on page 2a cannot be turned off.

Instantiating Megafunctions Using the Port & Parameter Definition

You can instantiate the megafunction directly in your AHDL, Verilog HDL, or VHDL code by calling the megafunction and setting its parameters as you would any other subdesign, module, or component.

Refer to the specific megafunction in the Quartus II Help for a list of the megafunction ports and parameters. Quartus II Help also provides a sample VHDL component declaration and AHDL function prototype for each megafunction.

Altera strongly recommends that you use the MegaWizard Plug-In Manager for complex megafunctions such as PLLs, transceivers, and LVDS drivers. For details on using the MegaWizard Plug-In Manager, refer to "Instantiating Megafunctions Using the MegaWizard Plug-In Manager" on page 7–3.

Inferring Altera Megafunctions from HDL Code

Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction. The synthesis tool uses the Altera megafunction code when compiling your design—even when you do not specifically instantiate the megafunction. Synthesis tools infer megafunctions to take advantage of logic that is optimized for Altera devices. The area and performance of such logic may be better than the results obtained by inferring generic logic from the same HDL code.

The following sections describe the types of logic that standard synthesis tools recognize and map to megafunctions. Synthesis software infers only the specific functions listed here. The software cannot infer other functions, such as PLLs, LVDS drivers, transceivers, or DDIO circuitry from HDL code. In some cases, you can use synthesis tool options to turn off inference of certain megafunctions. The following sections describe how to infer the following megafunctions from generic HDL code:

- "lpm_mult—Inferring Multipliers from HDL Code" on page 7–6
- "altmult_accum & altmult_add—Inferring Multiply-Accumulators & Multiply-Adders from HDL Code" on page 7–9
- "altsyncram & altdpram—Inferring RAM Functions from HDL Code" on page 7–12
- "lpm_rom—Inferring ROM from HDL Code" on page 7–24
- "altshift_taps—Inferring Shift Registers from HDL Code" on page 7–26

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*.

lpm_mult—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 multiplier 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 which functions it can implement, refer to the appropriate Altera device family handbook and The DSP Solution Center of the Altera web site at **www.altera.com**.

The following four code samples show Verilog HDL and VHDL examples for unsigned and signed multipliers that synthesis tools can infer as an `lpm_mult` or `altmult_add` megafunction. Each example fits into one DSP block 9-bit element. In addition, when register packing occurs, no extra logic cells for registers are required.

The `signed` declaration in Verilog HDL is a feature of the Verilog 2001 Standard.

Example 7–1. 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 7–2. Verilog HDL Signed Multiplier with Input & 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 7–3. VHDL Unsigned Multiplier with Input & 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 7–4. VHDL Signed Multiplier

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY signed_mult IS
   PORT (
      a: IN SIGNED (7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      result: OUT SIGNED (15 DOWNTO 0)
   );
END signed_mult;

BEGIN
   result <= a * b;
END rtl;
```

altmult_accum & altmult_add—Inferring Multiply-Accumulators & Multiply-Adders from HDL Code

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to `altmult_accum` or `altmult_add` megafunctions, respectively. 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 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 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.

The Verilog HDL and VHDL code samples shown in Examples 7–5 through 7–8 infer specific multiply-accumulators and multiply-adders.

Example 7–5. Verilog HDL Unsigned Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa;
   input [7:0] datab;
   input clk;
   input aclr;
   input clken;
   output [31:0] dataout;
   reg [31:0] dataout;
   reg [7:0] dataa_reg;
   reg [7:0] datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [31:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;
   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 8'b0;
         datab_reg <= 8'b0;
         multa_reg <= 16'b0;
         dataout <= 32'b0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

Example 7–6. Verilog HDL Signed Multiply-Adder (Latency = 0)

```
module sig_altmult_add (dataa, datab, datac, datad, result);
   input signed [15:0] dataa;
   input signed [15:0] datab;
   input signed [15:0] datac;
   input signed [15:0] datad;
   output [32:0] result;

   wire signed [31:0] mult0_result;
   wire signed [31:0] mult1_result;

   assign mult0_result = dataa * datab;
   assign mult1_result = datac * datad;
   assign result = (mult0_result + mult1_result);
endmodule
```

Example 7–7. VHDL Unsigned Multiply-Adder with Input, Output & Pipeline Registers (Latency = 3)

```
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;
```

Example 7–8. VHDL Signed Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```
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;
      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)
   BEGIN
      IF (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;
```

altsyncram & altdpram—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.

Synthesis tools recognize single-port and simple dual-port (one read port and one write port) RAM blocks. Tools usually do not infer small RAM blocks because small RAM blocks typically can be implemented more efficiently by 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. On the Assignments menu, click **Settings**. In the **Category** list, click **Analysis & Synthesis Settings.** Click **More Settings**. Under **Existing Options Settings**, select the option **Allow Any RAM Size for Recognition**. Click the **Setting** arrow and select **ON**.

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 potentially can cause compilation problems.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with TriMatrix™ memory blocks. For example, Quartus II integrated synthesis provides the `ramstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.

For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

When you are using a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules that contain only the RAM logic. In certain formal verification flows, for example, when using Quartus II integrated synthesis, the entity or module containing the inferred RAM is put into a black box automatically because formal verification tools do not support RAM blocks. The Quartus II software issues a warning message when this occurs. If the entity or module contains any additional logic outside the RAM block, this logic also must be treated as a black box for formal verification and therefore cannot be verified.

Single-Clock Synchronous RAM without Read-Through-Write Behavior

The code examples in this section show Verilog HDL and VHDL code that infers single-clock synchronous RAM. Altera's TriMatrix memory blocks are synchronous, so RAM designs targeting to architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture.

These examples also avoid read-through-write behavior, which is not directly supported in TriMatrix memory blocks. Altera recommends that you use this coding style as long as your design does not require RAM

with read-through-write behavior, meaning your design does not require that a simultaneous read and write to the same RAM location read the new value that is currently being written to that RAM location.

In TriMatrix memory blocks, if you attempt to read and write from the same address in the same clock cycle, the read returns either the old data at the address or unknown data, depending on the memory mode and block type.

If you require RAM with read-through-write behavior, refer to the section "Single-Clock Synchronous RAM with Read-Through-Write Behavior" on page 7–15.

For additional information about the dedicated memory blocks in your specific device, refer to the appropriate Altera device family data sheet on the Altera web site at **www.altera.com**.

The simple dual-port RAM code samples shown in Examples 7–9 and 7–10 map directly into Altera TriMatrix memory. Single-port versions of memory blocks (that is, using the same read address and write address signals) can allow better RAM utilization than dual-port memory blocks.

Example 7–9. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM without Read-Through-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 7–10. VHDL Single-Clock Simple Dual-Port Synchronous RAM Without Read-Through-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 Read-Through-Write Behavior

TriMatrix memory blocks do not support mixed-port read-through-write behavior. This means if you attempt to read and write from the same address in the same clock cycle, the read returns either the old data at the address or unknown data, depending on the memory mode and block type. However, you can describe a RAM block in HDL code in which a simultaneous read and write to the same location reads the new value that is currently being written to that RAM location.

The following examples show code that infers this type of RAM logic. 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.

The simple dual-port RAM examples shown in Examples 7–11 and 7–12 require 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 Stratix and Cyclone® series of devices, because the device memory supports read-through-write behavior when in single-port mode (same clock, same read and write address).

Example 7–11. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Read-Through-Write Behavior

```
module single_port_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
    end
endmodule
```

Note that Example 7–11 is similar to Example 7–9, but Example 7–11 uses non-blocking assignments so that the data is assigned immediately.

An alternative way to create a single-clock RAM is to use blocking assignments within the `always` block and add an assign statement to read the address of `mem` to create the output `q`. In this alternate coding style, if the memory block feeds a register with the same clock, old data exists in the register. You will not observe the desired read-through-write behavior. Synthesis tools may not infer a RAM block if this situation is possible in your design, such as the case when the memory feeds this kind of register through combinational logic or hard hierarchical partition boundaries. For this reason, Altera does not recommend using this alternate type of coding style.

Example 7–12. VHDL Single-Clock Simple Dual-Port Synchronous RAM with Read-through-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;
```

In the coding style shown in Example 7–12, if the memory block feeds a register with the same clock, old data exists in the register; therefore, you will not observe the desired read-through-write behavior. Synthesis tools may not infer a RAM block if this situation is possible in your design, such as the case when the memory feeds such a register through combinational logic or hard hierarchical partition boundaries.

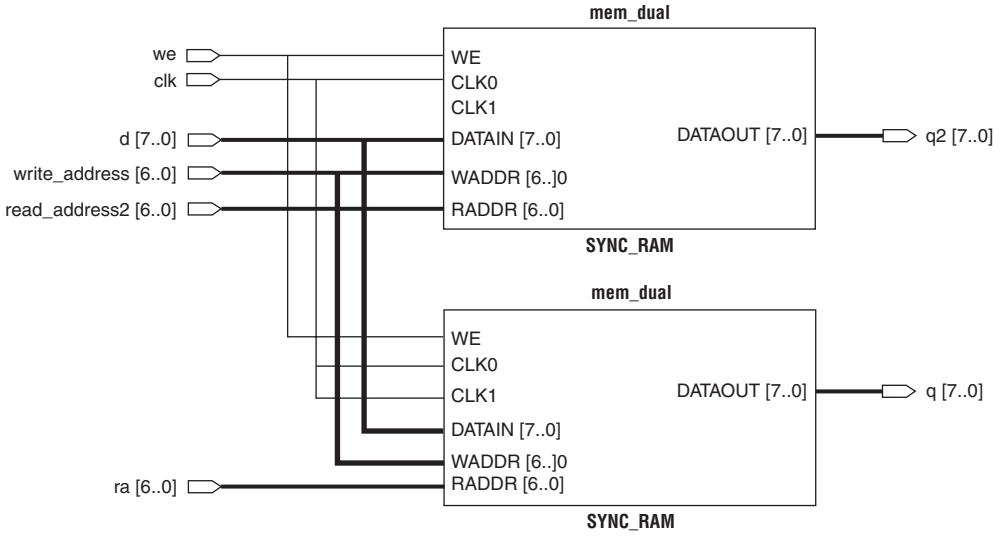
This example does not infer a RAM block for APEX devices by default because of the continuous read. For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM explicitly, rather than use the behavior specified by your HDL code.

For more information about 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*.

Synchronous RAM with Two Read Addresses

Quartus II integrated synthesis can infer RAM blocks from RAM descriptions that have two read addresses and one write address. This type of RAM blocks can be implemented by duplicating the RAM block as shown in Figure 7–1. All inputs are duplicated for both RAM blocks except for the read address, which is individual for each block.

Figure 7–1. Block Diagram Showing Synchronous RAM with Two Read Addresses



The RAM code samples with two read addresses shown in Examples 7–13 and 7–14 are inferred by duplicating the RAM block.

Example 7–13. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Two Read Addresses

```
module two_read_ram(
     output reg [7:0] q1, q2,
     input [7:0] d,
     input [6:0] write_address, read_address1, read_address2,
     input we, clk
);
     reg [7:0] mem [127:0];

     always @ (posedge clk) begin
          if (we)
               mem[write_address] <= d;
          q1 <= mem[read_address1];
          q2 <= mem[read_address2];
     end
endmodule
```

Example 7–14. VHDL Single-Clock Simple Dual-Port Synchronous RAM with Two Read Addresses

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY two_read_ram IS
     PORT (
          clock: IN STD_LOGIC;
          data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
          write_address: IN INTEGER RANGE 0 to 31;
          read_address1: IN INTEGER RANGE 0 to 31;
          read_address2: IN INTEGER RANGE 0 to 31;
          we: IN STD_LOGIC;
          q1: OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
          q2: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
     );
END two_read_ram;

ARCHITECTURE rtl OF two_read_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;
          q1 <= ram_block(read_address1);
          q2 <= ram_block(read_address2);
     END IF;
  END PROCESS;
END rtl;
```

Dual-Clock Synchronous RAM

Altera's TriMatrix memory blocks are synchronous, so RAM designs that target architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture. Synchronous memories are supported in all Altera device families.

When simultaneous reading and writing to the same RAM address occurs with two different clocks, the TriMatrix memory blocks in Altera devices return undefined data values. This usually differs from the functionality of the original HDL design. If your design requires a given output when reading and writing to the same RAM address, direct your synthesis tool not to infer RAM blocks for dual-clock memories by disabling RAM inference for these memories.

When Quartus II integrated synthesis infers this type of RAM, it issues a warning because of this undefined read-during-write cause. 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 explicitly, rather than use the behavior specified by your HDL code.

For specific options to disable RAM inference or read/write checks in 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 code samples shown in Examples 7–15 and 7–16 show Verilog HDL and VHDL code that infers dual-clock synchronous RAM.

Example 7–15. Verilog HDL 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 7–16. VHDL 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;
```

Specifying Initial Memory Contents

Your synthesis tool may offer various ways to specify the initial contents of an inferred memory.

For example, Quartus II integrated synthesis supports the `ram_init_file` synthesis attribute that allows you to specify a Memory Initialization File (**.mif**) for an inferred RAM block.

For information about the `ram_init_file` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the tool vendor's documentation.

In Verilog HDL, you can use an initial block to initialize the contents of an inferred memory. Quartus II integrated synthesis automatically converts the initial block into a MIF for the inferred RAM. Example 7–17 shows Verilog HDL code that infers a simple dual-port RAM block and corresponding MIF file.

Example 7–17. Verilog HDL RAM with Initialized Contents

```
module ram_with_init(
   output reg [7:0] q,
   input [7:0] d,
   input [4:0] write_address, read_address,
   input we, clk
);
   reg [7:0] mem [0:31];
   integer i;

   initial begin
      for (i = 0; i < 32; i = i + 1)
         mem[i] = i[7:0];
   end

   always @ (posedge clk) begin
      if (we)
         mem[write_address] <= d;
      q <= mem[read_address];
   end
endmodule
```

Quartus II integrated synthesis and other synthesis tools also support the `$readmemb` and `$readmemh` commands so that RAM and ROM initialization work identically in synthesis and simulation. Example 7–18 shows an initial block that initializes an inferred RAM block using the `$readmemb` command.

Example 7–18. 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 for the inferred RAM. Example 7–19 shows VHDL code that infers a simple dual-port RAM block and corresponding MIF file.

Example 7–19. 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;
```

lpm_rom—Inferring ROM from HDL Code

To infer ROM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `lpm_rom` megafunctions, depending on the target device family, only for device families that have dedicated memory blocks.

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.

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 are using the Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes 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**.
4. Under **Existing Options Settings**, select the option **Allow Any ROM Size for Recognition**.
5. Click the **Setting** arrow and select **ON**.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with TriMatrix memory blocks. For example, Quartus II integrated synthesis provides the `romstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.

For details about using the `romstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

When you are using a formal verification flow, Altera recommends that you create ROM blocks in separate entities or modules that contain only the ROM logic because you may need to treat the entity and module as a black box during formal verification.

The Verilog HDL and VHDL code samples shown in Examples 7–20 and 7–21 infer synchronous ROM blocks. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; consult the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Stratix series devices and newer device families, either the address or the output has to be registered for ROM code to be inferred. When output registers are used, the registers are implemented using the input registers of the RAM block, but the functionality of the ROM is not changed. If you register the address, the power-up state of the inferred ROM can be different from the HDL design. In this scenario, the synthesis software issues a warning. The Quartus II Help explains the condition under which the functionality changes when you are using Quartus II integrated synthesis.

These ROM code samples map directly to the Altera TriMatrix memory architecture.

Example 7–20. 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 7–21. 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;
```

altshift_taps—Inferring Shift Registers 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

Because formal verification tools do not support shift register megafunctions, the Quartus II integrated synthesis does not infer the `altshift_taps` megafunction when a formal verification tool is selected. You can select EDA tools for use with your Quartus II project on the **EDA Tool Settings** page of the **Settings** dialog box.

When you are using 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 may need to treat the entity or module as a black box during formal verification.

Synthesis software recognizes shift registers only for device families that have dedicated RAM blocks and the software uses certain guidelines to determine the best implementation. The following guidelines are followed in Quartus II integrated synthesis and also are generally followed by other EDA tools:

- For FLEX® 10K and ACEX® 1K devices, the software does not infer `altshift_taps` megafunctions because FLEX 10K and ACEX 1K devices have a relatively small amount of dedicated memory.

- For APEX 20K and APEX II devices, the software infers the `altshift_taps` megafunction only if the shift register has more than a total of 128 bits. Smaller shift registers typically do not benefit from implementation in dedicated memory.
- For Stratix and Cyclone series devices, the software determines whether to infer the `altshift_taps` megafunction based on the width of the registered bus (*W*), the length between each tap (*L*), and the number of taps (*N*).
  - If the registered bus width is one ($W = 1$), the software infers `altshift_taps` if the number of taps times the length between each tap is greater than or equal to 64 ($N \times L \geq 64$).
  - If the registered bus width is greater than one ($W > 1$), the software infers `altshift_taps` if the registered bus width times the number of taps times the length between each tap is greater than or equal to 32 ($W \times N \times L \geq 12$).

If the length between each tap (*L*) is not a power of two, the software uses more logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or Adaptive Logic Modules (ALMs) is required to implement the function. This decode logic eliminates the performance and utilization advantages of implementing shift registers in memory.

The registers that the software maps to the `altshift_taps` megafunction and places in RAM are not available in a Verilog HDL or VHDL output file for simulation tools because their node names do not exist after synthesis.

Simple Shift Register

The code sample shown in Example 7–22 and Example 7–23 show a simple, single-bit wide, 64-bit long shift register. The synthesis software implements the register ($W = 1$ and $M = 64$) in an `altshift_taps` megafunction for supported devices. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 7–22. 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 7–23. VHDL Single-Bit Wide, 64-Bit Long Shift Register

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_1x64 IS
   PORT (
      clk: IN STD_LOGIC;
      shift: IN STD_LOGIC;
      sr_in: IN STD_LOGIC;
      sr_out: OUT STD_LOGIC
   );
   END shift_1x64;

ARCHITECTURE arch OF shift_1x64 IS
   TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
   SIGNAL sr: sr_length;
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT and clk = '1') THEN
         IF (shift = '1') THEN
            sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
            sr(0) <= sr_in;
         END IF;
      END IF;
   END PROCESS;
   sr_out <= sr(63);
END arch;
```

Shift Register with Evenly Spaced Taps

The code samples shown in Examples 7–24 and 7–25 show a Verilog HDL and VHDL 8-bit wide, 64-bit long shift register ($W > 1$ and $M = 64$) with evenly spaced taps at 15, 31, and 47. The synthesis software implements this function in a single `altshift_taps` megafunction and maps it to RAM in supported devices.

Example 7–24. 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 7–25. 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 & 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 provides 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 & Clock Enable" on page 7–33
- "Latches" on page 7–37

Register Power-Up Values in Altera Devices

Registers in the device core always power up to a low (0) logic level on all Altera devices. However, there are ways to implement logic such that registers behave as if they were powering up to a high (1) logic level.

If you use a preset signal on a device that does not support presets in the register architecture, then your synthesis tool may convert the preset signal to a clear signal, which requires synthesis to perform an optimization referred to as NOT gate push-back. NOT gate push-back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high but the device operates as expected. In this case, your synthesis tool may issue a message informing you about the power-up condition. The register itself powers up low, but the register output is inverted so the signal that arrives at all destinations is high.

Due to these effects, if you specify a particular reset value (other than 0), you may cause your synthesis tool to use the asynchronous clear (aclr) signals available on the registers to implement the high bits with NOT gate push-back. In that case, the registers look as though they power up to the specified reset value. You see this behavior, for example, if your design targets FLEX 10KE or ACEX devices.

When a load signal is available in the device, your synthesis tools can implement a reset of 1 or 0 value by using an asynchronous load of 1 or 0. When the synthesis tool uses an asynchronous load signal, it is not performing NOT gate push-back, so the registers power up to a 0 logic level.

For additional details, refer to the appropriate device family handbook or the appropriate handbook of the Altera web site at www.altera.com.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset but not necessarily at power-up. You can create your design such that the asynchronous reset allows the board to operate in a safe condition and then you can bring up the design with the reset active. This is a good practice so you do not depend on the power-up conditions of the device.

You can make the your design more stable and avoid potential glitches by synchronizing external or combinational logic of the device architecture before you drive the asynchronous control ports of registers.

For additional information about good synchronous design practices, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

If you want to force a particular power-up condition for your design, use the synthesis options available in your synthesis tool. With Quartus II integrated synthesis, you can apply the **Power-Up Level** logic option. You can also apply the option with an `altera_attribute` assignment in your source code. Using this option forces synthesis to perform `NOT` gate push-back because synthesis tools cannot actually change the power-up states of core registers.

You can apply the Quartus II integrated synthesis **Power-Up Level** assignment to a specific register or to a design entity, module or subdesign. If you do so, every register in that block receives the value. Registers power up to 0 by default; therefore you can use this assignment to force all registers to power up to 1 using `NOT` gate push-back.

Be aware that using `NOT` gate push-back as a global assignment could slightly degrade the quality of results due to the number of inverters that are needed. In some situations, issues are caused by enable or secondary control logic inference. It may also be more difficult to migrate such a design to an ASIC or a HardCopy® device. You can simulate the power-up behavior in a functional simulation if you use initialization.

The **Power-Up Level** option and the `altera_attribute` assignment are described in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Some synthesis tools can also read the default or initial values for registered signals and implement this behavior in the device. For example, Quartus II integrated synthesis converts default values for registered signals into Power-Up Level settings. That way, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

For example, the code in Example 7–26 infers a register for `q` and sets its power-up level to high (while the reset value is `0`):

Example 7–26. Inferring a Register for q & Setting Power-Up Level to High

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
   IF (reset = '1') THEN
      q <= '0';
   ELSIF (rising_edge(clk)) THEN
      q <= d;
   END IF;
END PROCESS;
```

Secondary Register Control Signals Such as Clear & Clock Enable

FPGA device architectures contain registers, also known as "flipflops". The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells. Device families vary in their support for secondary signals, so consult the device family data sheet to verify which signals are available in your target device.

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. The control signals have a certain priority due to the nature of the architecture, so your HDL code should follow that priority where possible.

Your synthesis tool can emulate any control signals using regular logic, so getting functionally correct results is always possible. However, if your design requirements are flexible in terms of which control signals are used and in what priority, match your design to the target device architecture to achieve the most efficient results. If the priority of the signals in your design is not the same as that of the target architecture, then extra logic may be required to implement the control signals. This extra logic uses additional device resources, and can cause additional delays for the control signals.

In addition, there are certain cases where using logic other than the dedicated control logic in the device architecture can have a larger impact. For example, the clock enable signal has priority over the synchronous reset or clear signal in the device architecture. The clock enable turns off the clock line in the logic array block (LAB), and the clear signal is synchronous. So in the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a synchronous clear signal that has priority over the clock enable signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the clock enable port of a register, you cannot apply a Clock Enable Multicycle constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals, and using a different priority causes unexpected results with an assignment to the clock enable signal.

The priority order for secondary control signals in Altera devices differs from the order for other vendors' devices. If your design requirements are flexible regarding priority, verify that the secondary control signals meet design performance requirements when migrating designs between FPGA vendors and try to match your target device architecture to achieve the best results.

The signal order is the same for all Altera device families, although as noted previously, not all device families provide every signal. The following priority order is observed:

1. Asynchronous Clear, `aclr`—highest priority
2. Preset, `pre`
3. Asynchronous Load, `aload`
4. Enable, `ena`
5. Synchronous Clear, `sclr`
6. Synchronous Load, `sload`
7. Data In, `data`—lowest priority

The following examples provide Verilog HDL and VHDL code that creates a register with the `aclr`, `aload`, and `ena` control signals.

The Verilog HDL example (Example 7–27) does not have `adata` on the sensitivity list, but the VHDL example (Example 7–28) 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 7–27. Verilog HDL D-Type Flipflop (Register) with ena, aclr & 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 7–28. VHDL D-Type Flipflop (Register) with ena, aclr & 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, because it is replaced with the more flexible aload signal, so the preset signal 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.

Therefore, synthesis tools typically avoid using the sload or sclear signal if there is space in the look-up table (LUT). Using the LUT to implement the signals is always more flexible if it is available.

Synthesis tools also typically restrict use of sload and sclr signals to certain functions such as arithmetic chains (counters), or wide multiplexers in which there are enough registers with common signals to allow good LAB packing. Because different device families offer different numbers of control signals, inference of these signals is also device-specific. For example, Stratix II devices have more flexibility than Stratix devices with respect to secondary control signals, so synthesis tools might infer more sload and sclr signals for Stratix II devices.

If you use these additional control signals, use them in the priority order that matches the device architecture. To achieve the most efficient results, ensure the sclr signal has a higher priority than the sload signal in the same way that aclr has higher priority than aload in the previous examples. Remember that the register signals are not inferred unless the design meets the conditions described previously. However, if your HDL described the desired behavior, the software always implements logic with the correct functionality.

In Verilog HDL, the following code for sload and sclr could replace the if (ena) q <= data; statements in the Verilog HDL example shown in Example 7–27 on page 7–35 (after adding the control signals to the module declaration).

Example 7–29. Verilog HDL sload & sclr Control Signals

```
if  (ena) begin
   if  (sclr)
      q <= 1'b0;
   else if  (sload)
      q <= sdata;
   else
      q <= data;
end
```

In VHDL, the following code for sload and sclr could replace the IF (ena ='1') THEN q <= data; END IF; statements in the VHDL example shown in Example 7–28 on page 7–35 (after adding the control signals to the entity declaration).

Example 7–30. VHDL sload & 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 all combinational loops, refer to the *Design Recommendations for Altera Devices* 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 detailed 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 7–39.

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 provides an example explaining possible simulation mismatches.

Omitting the final ELSE or WHEN OTHERS clause in an IF or CASE statement can also generate a latch. Don't care (X) assignments on the default conditions are useful in preventing latch generation. For the best logic optimization, assign the default CASE or final ELSE value to don't care (X) instead of a logic value.

The VHDL sample code shown in Example 7–31 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 7–31. 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 problems typically associated with combinational loops.

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 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** report, 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 will never encounter data that causes that path to be activated, or the surrounding logic is set up in a mutually exclusive manner that prevents that path from ever being sensitized, independent of the data input.

For macrocell-based devices such as MAX® 7000AE and MAX 3000A, all data (D-type) latches and set-reset (S-R) latches listed in the **Analysis & Synthesis User-Specified and Inferred Latches** table have an implementation free of timing hazards such as glitches. The implementation includes a cover term to ensure there is no glitching, and includes a single macrocell in the feedback loop.

For 4-input LUT-based devices such as Stratix devices, the Cyclone series, and MAX II devices, all latches in the **User-Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value. For example, a D-type input toggling when the enable input is inactive, or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is needed 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, then the **User-Specified and Inferred Latches** table indicates that the latch is not free of timing hazards.

For 6-input LUT-based devices, the software can implement all latch inputs with a single adaptive look-up table (ALUT) in the combinational loop. Therefore, all latches in the **User-Specified and Inferred Latches** table are free of timing hazards when a single input changes.

If a latch is listed as a safe latch, other Quartus II optimizations, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance.

To ensure hazard-free behavior, only one control input may change at a time. Changing two inputs simultaneously, such as deasserting set and reset at the same time, or changing data and enable at the same time, can produce incorrect behavior in any latch.

Quartus II integrated synthesis infers latches from `always` blocks in Verilog HDL and `process` statements in VHDL, but not from continuous assignments in Verilog HDL or concurrent signal assignments in VHDL. These rules are the same as for register inference. The software infers registers or flipflops only from `always` blocks and `process` statements.

The Verilog HDL code sample shown in Example 7–32 infers a S-R latch correctly in the Quartus II software.

Example 7–32. Verilog HDL Set-Reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1
      else if (ResetTerm)
         LatchOut = 1'b0
   end
endmodule
```

The VHDL code sample shown in Example 7–33 infers a D-type latch correctly in the Quartus II software.

Example 7–33. VHDL Data Type Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY simple_latch IS
   PORT (
      enable, data    : IN STD_LOGIC;
      q               : OUT STD_LOGIC
   );
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

latch : PROCESS (enable, data)
   BEGIN
      IF (enable = '1') THEN
         q <= data;
      END IF;
   END PROCESS latch;
END rtl;
```

The following example shows a Verilog HDL continuous assignment that does not infer a latch in the Quartus II software. The behavior is similar to a latch, but it may not function correctly as a latch and its timing is not analyzed as a latch.

```
assign latch_out = (~en & latch_out) | (en & data);
```

Quartus II integrated synthesis also creates safe latches when possible for instantiations of the `lpm_latch` megafunction. You can use this megafunction to create a latch with any combination of data, enable, set, and reset inputs. The same limitations apply for creating safe latches as for inferring latches from HDL code.

Inferring the Altera `lpm_latch` function in another synthesis tool ensures that the implementation is also recognized as a latch in the Quartus II software. If a third-party synthesis tool implements a latch using the `lpm_latch` megafunction, then the Quartus II integrated synthesis lists the latch in the **User-Specified and Inferred Latches** table in the same way as it lists latches created in HDL source code. The coding style necessary to produce an `lpm_latch` implementation may depend on your synthesis tool. Some third-party synthesis tools list the number of `lpm_latch` functions that are inferred.

For LUT-based families, the Fitter uses global routing for control signals including signals that Analysis & 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.
- "Adder Trees" on page 7–44. This section explains the different coding styles that lead to optimal results for devices with 4-input look-up tables and 6-input adaptive look-up tables.
- "State Machines" on page 7–46. This section helps ensure the best results when you use state machines.
- "Multiplexers" on page 7–53. 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 7–63. This section provides guidelines for getting good results when designing CRC functions.

- "Comparators" on page 7–65. This section explains different comparator implementations and provides suggestions for controlling the implementation.
- "Counters" on page 7–67. This section provides guidelines to ensure your counter design targets the device architecture optimally.

Tri-State Signals

When you are targeting Altera devices, you should use tri-state signals only when they are attached to top-level bidirectional or output pins. Avoid lower level bidirectional pins, and avoid using the `Z` logic value unless it is driving an output or bidirectional pin.

Synthesis tools implement designs with internal tri-state signals correctly in Altera devices using multiplexer logic, but Altera does not recommend this coding practice.

In hierarchical block-based or incremental design flows, a hierarchical boundary cannot contain any bidirectional ports, unless the lower level bidirectional port is connected directly through the hierarchy to a top-level output pin without connecting to any other design logic. If you use boundary tri-states in a lower level block, synthesis software must push the tri-states through the hierarchy to the top-level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower level tri-states are restricted with block-based design methodologies.

The code examples shown in Examples 7–34 and 7–35 show Verilog HDL and VHDL code that creates tri-state bidirectional signals.

Example 7–34. Verilog HDL Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;
    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Example 7–35. 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;
```

Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can result in significant performance and density improvements. A good example of an application using a large adder tree is a finite impulse response (FIR) correlator. Using a pipelined binary or ternary adder tree appropriately can greatly improve the quality of your results.

This section explains why coding recommendations are different for Altera 4-input LUT devices and 6-input LUT devices.

Architectures with 4-Input LUTs in Logic Elements

Architectures such as Stratix devices and the Cyclone series, APEX series, and FLEX series 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.

In the code sample shown in Example 7–36, 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 7–36. 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

Newer 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 the previous example 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 look-up table (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 7–37 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 7–37. 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. In some designs that are operated at high temperatures, designers are also concerned that alpha particles could change the state of the circuit.

Of course this option protects only state machines, and all other registers in the design are not protected this way.

For additional information about tool-specific options for implementing state machines, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The following two sections, "Verilog HDL State Machines" and "VHDL State Machines" on page 7–51, 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 you are using the SystemVerilog standard, use enumerated types to describe state machines (as shown in the "SystemVerilog State Machine Coding Example" on page 7–51).
- Represent the states in a state machine with the `parameter` data types in Verilog-1995 and -2001 and use the parameters to make state assignments (as shown below in the "Verilog HDL State Machine Coding Example"). This implementation makes the state machine easier to read and reduces the risk of errors during coding.

  Altera recommends against the direct use of integer values for state variables such as `next_state <= 0`. However, using an integer does not prevent inference in the Quartus II software.

- No state machine is inferred in the Quartus II software if the state transition logic uses arithmetic similar to that shown in the following example:

```
case (state)
    0: begin
        if (ena) next_state <= state + 2;
        else next_state <= state + 1;
    end
    1: begin
    ...
endcase
```

- No state machine is inferred in the Quartus II software if the state variable is an output.

Verilog HDL State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation (Example 7–38).

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 7–38. Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk;
    input reset;
    input [3:0] in_1;
    input [3:0] in_2;output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
            state <= next_state;
    end
    always @ (state or in_1 or in_2)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
```

```
            state_0: begin
               tmp_out_2 <= in_1 + 5'b00001;
               next_state <= state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state <= state_2;
                    tmp_out_2 <= tmp_out_0;
                end
                else begin
                    next_state <= state_3;
                    tmp_out_2 <= tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 <= tmp_out_0 - 5'b00001;
                next_state <= state_3;
            end
            state_3: begin
                tmp_out_2 <= tmp_out_1 + 5'b00001;
                next_state <= state_0;
            end
            state_4:begin
                tmp_out_2 <= in_2 + 5'b00001;
                next_state <= state_0;
            end
            default:begin
                tmp_out_2 <= 5'b00000;
                next_state <= state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `‘define` instead of the `parameter` data type, as follows:

```
‘define state_0 3'b000
‘define state_1 3'b001
‘define state_2 3'b010
‘define state_3 3'b011
‘define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a `‘state_x` instead of a `state_x`, as shown in the following example:

```
next_state <= ‘state_3;
```

Although the `‘define` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

SystemVerilog State Machine Coding Example

The module `enum_fsm` shown in Example 7–39 is an example of a SystemVerilog state machine implementation that uses enumerated types. Altera recommends using this coding style to describe state machines in SystemVerilog.

In Quartus II integrated synthesis, the enumerated type that defines the states for the state machine must be of an unsigned integer type as shown in Example 7–39. 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 recognize or infer a state machine.

Example 7–39. 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 Compilation Report.

VHDL State Machine Coding Example

The following entity, `vhdl_fsm`, is an example of a typical VHDL state machine implementation (Example 7–40).

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 7–40. VHDL State Machine

```
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);
   BEGIN
      tmp_out_0 := in1 + in2;
```

```
      tmp_out_1 := in1 - in2;
      CASE state IS
         WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
         WHEN state_1 =>
            IF (in1 < in2) then
               next_state <= state_2;
               out_1 <= tmp_out_0;
            ELSE
               next_state <= state_3;
               out_1 <= tmp_out_1;
            END IF;
         WHEN state_2 =>
            IF (in1 < "0100") then
               out_1 <= tmp_out_0;
            ELSE
               out_1 <= tmp_out_1;
            END IF;
               next_state <= state_3;
         WHEN state_3 =>
               out_1 <= "11111";
               next_state <= state_4;
         WHEN state_4 =>
               out_1 <= in2;
               next_state <= state_0;
         WHEN OTHERS =>
               out_1 <= "00000";
               next_state <= state_0;
      END CASE;
   END PROCESS;
END rtl;
```

Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexer logic, you ensure the most efficient implementation in your Altera device. This section addresses common problems and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes various types of multiplexers, and how they are implemented in the 4-input LUT found in many FPGA architectures, such as Altera's Stratix devices.

Stratix II and newer high-performance devices have 6-input ALUTs and are not specifically addressed here. Although many of the principles and techniques for optimization are similar, device utilization differs in the 6-input LUT devices. For example, these devices can implement wider multiplexers in one ALM than can be implemented in the 4-input LUT of an LE.

Quartus II Software Option for Multiplexer Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that extracts and optimizes buses of multiplexers during synthesis. In certain situations, this option automatically performs some of the recoding functions described in this section without changing the HDL code in your design. This option is on by default, when the Optimization technique is set to **Balanced** (the default for most device families) or set to **Area**.

For details, refer to the *Restructure Multiplexers* subsection in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Even with this Quartus II-specific option turned on, it is beneficial to understand how your coding style can be interpreted by your synthesis tool, and avoid the situations that can cause problems in your design.

Multiplexer Types

This first subsection addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs. These HDL structures create different types of multiplexers including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code and how they might be implemented during synthesis is the first step towards optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 7–41 shows Verilog HDL code describing a simple 4:1 binary multiplexer.

Example 7–41. Verilog HDL Binary-Encoded Case Statement

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

A 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers can be constructed that use the 4:1 multiplexer; constructing an *N*-input multiplexer (*N*:1 multiplexer) from a tree of 4:1 multiplexers can result in a structure using as few as 0.66*(*N* - 1) LUTs.

Selector Multiplexers

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 7–42 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 7–42. Verilog HDL One-Hot-Encoded Case Statement

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

Selector multiplexers are commonly built as a tree of AND and OR gates. Using this scheme, two inputs can be selected using two select lines in a single 4-input LUT that uses two AND gates and an OR gate. The outputs of these LUTs can be combined with a wide OR gate. An *N*-input selector multiplexer of this structure requires at least 0.66*(*N*-0.5) LUTs, which is just slightly worse than the best binary multiplexer.

Priority Multiplexers

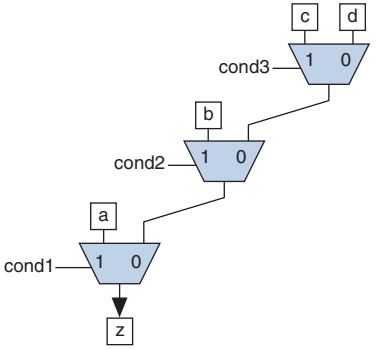
In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from IF, ELSE, WHEN, SELECT, and ?: statements in VHDL or Verilog HDL. The example VHDL code in Example 7–43 will probably result in the schematic implementation illustrated in Figure 7–2.

Example 7–43. VHDL IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

The multiplexers shown in Figure 7–2 form a chain, evaluating each condition or select bit, one at a time.

Figure 7–2. Priority Multiplexer Implementation of an IF Statement



An *N*-input priority multiplexer uses a LUT for every 2:1 multiplexer in the chain, requiring *N*-1 LUTs. This chain of multiplexers generally increases delay because the critical path through the logic traverses every multiplexer in the chain.

To improve the timing delay through the multiplexer, avoid priority multiplexers if priority is not required. If the order of the choices is not important to the design, use a `CASE` statement to implement a binary or selector multiplexer instead of a priority multiplexer. If delay through the structure is important in a multiplexed design requiring priority, consider recoding the design to reduce the number of logic levels to minimize delay, especially along your critical paths.

Default or Others Case Assignment

To fully specify the cases in a `CASE` statement, include a `DEFAULT` (Verilog HDL) or `OTHERS` (VHDL) assignment. This assignment is especially important in one-hot encoding schemes where many combinations of the select lines are unused. Specifying a case for the unused select line combinations gives the synthesis tool information about how to synthesize these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not require that the outcome in the unused cases be considered, often because designers assume these cases will not occur. For these types of designs, you can choose any value for the `DEFAULT` or `OTHERS` assignment. However, be aware that the assignment value you choose can have a large effect on the logic utilization required to implement the design due to the different ways synthesis tools treat different values for the assignment, and how the synthesis tools use different speed and area optimizations.

In general, to obtain best results, explicitly define invalid CASE selections with a separate DEFAULT or OTHERS statement instead of combining the invalid cases with one of the defined cases.

If the value in the invalid cases is not important, specify those cases explicitly by assigning the X (don't care) logic value instead of choosing another value. This assignment allows your synthesis tool to perform the best area optimizations.

You can experiment with different DEFAULT or OTHERS assignments for your HDL design and your synthesis tool to test the effect they have on logic utilization in your design.

Implicit Defaults

The IF statements in Verilog HDL and VHDL can be a convenient way to specify conditions that do not easily lend themselves to a CASE-type approach. However, using IF statements can result in complicated multiplexer trees that are not easy for synthesis tools to optimize.

In particular, every IF statement has an implicit ELSE condition, even when it is not specified. These implicit defaults can cause additional complexity in a multiplexed design.

The code in Example 7–44 represents a multiplexer with four inputs (a, b, c, d) and one output (z).

Example 7–44. VHDL IF Statement with Implicit Defaults

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   END IF;
ELSIF cond6 THEN
   z <= d;
END IF;
```

This is not a recommended coding style. Although the code appears to implement a 4:1 multiplexer, each of the three separate IF statements in the code has an implicit ELSE condition that is not specified. Because the output values for the ELSE cases are not specified, the synthesis tool assumes the intent is to maintain the same output value for these cases

and infers a combinational loop, such as a latch. Latches add to the design's logic utilization and can also make timing analysis difficult and lead to other problems.

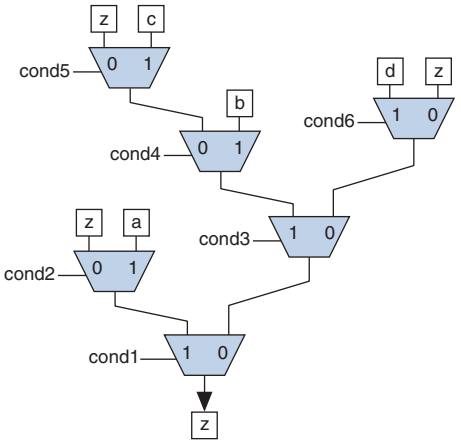
The code sample shown in Example 7–45 shows code with the same functionality as the code shown in Example 7–44, but specifies the ELSE cases explicitly. (This is not a recommended coding style improvement, but it explicitly shows the default conditions from the previous example.)

Example 7–45. VHDL IF Statement with Default Conditions Explicitly Specified

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   ELSE
      z <= z;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   ELSE
      z <= z;
   END IF;
ELSIF cond6 THEN
   z <= d;
ELSE
   z <= z;
END IF;
```

Figure 7–3 is a schematic representing the code in Example 7–45, which illustrates that the multiplexer logic is significantly more complicated than a basic 4:1 multiplexer, although there are only four inputs.

Figure 7–3. Multiplexer Implementation of an IF Statement with Implicit Defaults



There are several ways you can simplify the multiplexed logic and remove the 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 deduce default cases and flatten the multiplexer. In this example, instead of `IF cond1 THEN IF cond2`, use `IF (cond1 AND cond2)`, which performs the same function. In addition, examine whether the defaults are don't care cases. In this example, you can promote the last `ELSIF cond6` statement to an `ELSE` statement if no other valid cases can occur.

Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

Degenerate Multiplexers

A degenerate multiplexer is a multiplexer in which not all of the possible cases are used for unique data inputs. The unneeded cases tend to contribute to inefficiency in the logic utilization for these multiplexers. You can recode degenerate multiplexers so they take advantage of the efficient logic utilization possible with full binary multiplexers.

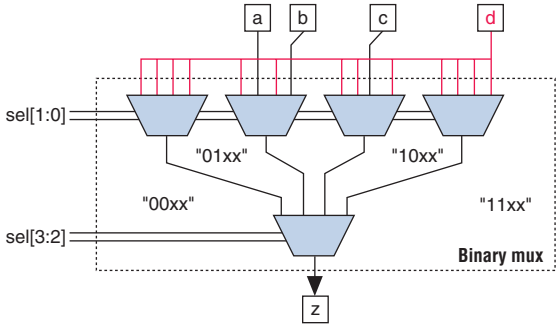
Example 7–46 shows a VHDL CASE statement describing a degenerate multiplexer.

Example 7–46. VHDL CASE Statement Describing a Degenerate Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z <= a;
   WHEN "0111" => z <= b;
   WHEN "1010" => z <= c;
   WHEN OTHERS => z <= d;
END CASE;
```

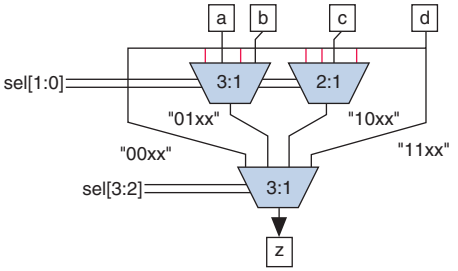
The number of select lines in a binary multiplexer normally dictates the size of a multiplexer needed to implement the desired function. For example, the multiplexer structure represented in Figure 7–4 has four select lines capable of implementing a binary multiplexer with 16 inputs. However, the design does not use all 16 inputs, which makes this multiplexer a degenerate 16:1 multiplexer.

Figure 7–4. Binary Degenerate Multiplexer



In the example in Figure 7–4, the first and fourth multiplexers in the top level can easily be eliminated because all four inputs to each multiplexer are the same value, and the number of inputs to the other multiplexers can be reduced, as shown in Figure 7–5.

Figure 7–5. Optimized Version of the Degenerate Binary Multiplexer



Implementing this version of the multiplexer still requires at least five 4-input LUTs, two for each of the remaining 3:1 multiplexers and one for the 2:1 multiplexer. This design selects an output from only four inputs, a 4:1 binary multiplexer can be implemented optimally in two LUTs, so this degenerate multiplexer tree reduces the efficiency of the logic.

You can improve logic utilization of this structure by recoding the select lines to implement a full 4:1 binary multiplexer. The code sample shown in Example 7–47 shows a recoder design that translates the original select lines into a signal `z_sel` with binary encoding.

Example 7–47. VHDL Recoder Design for Degenerate Binary Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z_sel <= "00";
   WHEN "0111" => z_sel <= "01";
   WHEN "1010" => z_sel <= "10";
   WHEN OTHERS => z_sel <= "11";
END CASE;
```

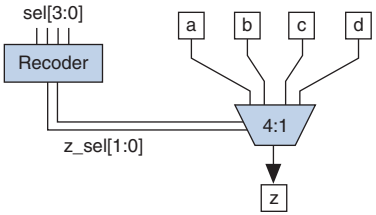
The code sample shown in Example 7–48 shows you how to implement the full binary multiplexer.

Example 7–48. VHDL 4:1 Binary Multiplexer Design

```
CASE z_sel[1:0] IS
   WHEN "00" => z <= a;
   WHEN "01" => z <= b;
   WHEN "10" => z <= c;
   WHEN "11" => z <= d;
END CASE;
```

Use the new `z_sel` control signal from the recoder design to control the 4:1 binary multiplexer that chooses between the four inputs `a`, `b`, `c`, and `d`, as illustrated in Figure 7–6. The complexity of the select lines is handled in the recoder design, and the data multiplexing is performed with simple binary select lines enabling the most efficient implementation.

Figure 7–6. Binary Multiplexer with Recorder



The design for the recoder can be implemented in two LUTs and the efficient 4:1 binary multiplexer uses two LUTs, for a total of four LUTs. The original degenerate multiplexer required five LUTs, so the recoded version uses 20% less logic than the original.

You can often improve the logic utilization of multiplexers by recoding the select lines into full binary cases. Although logic is required to perform the encoding, the overall logic utilization is often improved.

Buses of Multiplexers

The inputs to multiplexers are often data input buses in which the same multiplexer function is performed on a set of data input buses. In these cases, any inefficiency in the multiplexer is multiplied by the number of bits in the bus. The issues described in the previous sections become even more important for wide multiplexer buses.

For example, the recoding of select lines into full binary cases detailed in the previous section can often be used in multiplexed buses. Recoding the select lines may need to be completed only once for all the multiplexers in the bus. By sharing the recoder logic among all the bits in the bus, you can greatly improve the logic efficiency of a bus of multiplexers.

The degenerate multiplexer in the previous section requires five LUTs to implement. If the inputs and output are 32 bits wide, the function could require 32 × 5 or 160 LUTs for the whole bus. The recoder design uses only two LUTs, and the select lines only need to be recoded once for the entire bus. The binary 4:1 multiplexer requires two LEs per bit of the bus. The

total logic utilization for the recoded version could be 2 + (2 x 32) or 66 LUTs for the whole bus, compared to 160 LUTs for the original version. The logic savings become more important with wide multiplexer buses.

Using techniques to optimize degenerate multiplexers, removing unneeded implicit defaults, and choosing the optimal `DEFAULT` or `OTHERS` case can play an important role when optimizing buses of multiplexers.

Cyclic Redundancy Check Functions

Cyclic redundancy check (CRC) computations are used heavily by communications protocols and storage devices to detect any corruption of the data. These functions are highly effective; there is a very low probability that corrupted data can pass a 32-bit CRC check.

CRC functions typically use wide `XOR` gates to compare the data. The way that synthesis tools flatten and factor these `XOR` gates to implement the logic in FPGA LUTs can greatly impact the area and performance results for the design. `XOR` gates have a cancellation property which creates an exceptionally large number of reasonable factoring combinations, so synthesis tools cannot always choose the best result by default.

The 6-input ALUT has a significant advantage over 4-input LUTs for these designs. When properly synthesized, CRC processing designs can run at high speeds in devices with 6-input ALUTs.

The following guidelines help you improve the quality of results for CRC designs in Altera devices.

If Performance is Important, Optimize for Speed

Synthesis tools flatten `XOR` gates to minimize area and depth of levels of logic. Synthesis tools such as Quartus II integrated synthesis target area optimization by default for these logic structures. Therefore, for more focus on depth reduction, set the synthesis optimization technique to speed.

Note that 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 needed (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 need in the design, then multiplex them together to choose the appropriate mode at a given time.

Use Separate CRC Blocks Instead of Allowing Blocks to Merge

Synthesis tools often attempt to optimize CRC designs by sharing resources and extracting duplicates in two different CRC blocks because of the factoring options in the XOR logic. As addressed previously, the CRC logic allows significant reductions but this works best when each CRC function is optimized separately. Check for duplicate extraction behavior if you have different CRC functions that are driven by common data signals or that feed the same destination signals.

If you are having problems with the quality of results and you see that two CRC functions are sharing logic, ensure that the blocks are synthesized independently using one of the following methods:

- Define each CRC block as a separate design partition in an incremental compilation design flow.

  For details, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

- Synthesize each CRC block as a separate project and then write a separate VQM or EDIF netlist file for each.
  - To create a VQM file using Quartus II integrated synthesis, on the Processing menu, point to Start and click **Start VQM Writer**.

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 needed. 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 & Clock Enable" on page 7–33. You can check the register equations in the Timing Closure Floorplan or the Chip Planner to ensure that the signal was used as expected.

If you must force a register implementation using an `sload` signal, you can use low-level device primitives as described in the *Introduction to Low-Level Primitives Design User Guide*.

Comparators

Synthesis software, including Quartus II integrated synthesis, uses device and context-specific implementation rules for comparators (<, >, or ==) and selects the best one for your design. This section provides some information about the different types of implementations available and provides suggestions on how you can code your design to encourage a specific implementation.

The == comparator is implemented in general logic cells. The < comparison can be implemented using the carry chain or general logic cells. In devices with 6-input ALUTs, the carry chain is capable of comparing up to three bits per cell. In devices with 4-input LUTs, the capacity is one bit of comparison per cell, similar to an add/subtract chain. The carry chain implementation tends to be faster than the general logic on standalone benchmark test cases, but can result in lower performance when it is part of a larger design due to the increased restriction on the Fitter. The area requirement is similar for most input patterns. The synthesis software selects an appropriate implementation based on the input pattern.

If you are using Quartus II integrated synthesis, you can guide the synthesis by using specific coding styles. To select a carry chain implementation explicitly, rephrase your comparison in terms of addition. As a simple example, the following coding style allows the synthesis tool to select the implementation, which is most likely using general logic cells in modern device families:

```
wire [6:0] a,b;
wire alb = a<b;
```

In the following coding style, the synthesis tool uses a carry chain (except for a few cases, such as when the chain is very short or the signals `a` and `b` minimize to the same signal):

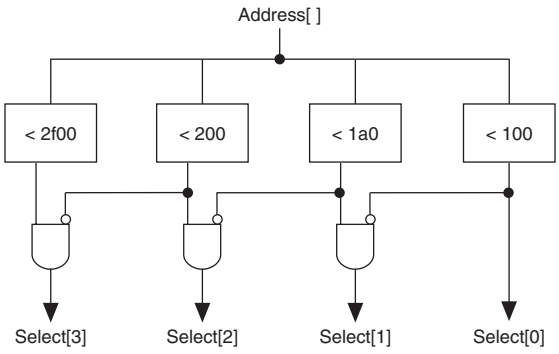
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

This second coding style uses the top bit of the `tmp` signal, which is 1 in twos complement logic if *a* is less than *b*, because the subtraction *a* – *b* results in a negative number.

If you have any information about the range of the input, you have "don't care" values that you can use to optimize the design. Because this information is not available to the synthesis tool, you can often reduce the device area required to implement the comparator with specific hand implementation of the logic.

You can also check whether a bus value is within a constant range with a small amount of logic area by using the logic structure shown in Figure 7–7. This type of logic occurs frequently in address decoders.

Figure 7–7. 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 look-up table 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

Refer to the *Designing With Low-Level Primitives User Guide* for details about and examples of using these types of assignments.

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 & Timing Optimization* chapter in volume 2 of the *Quartus II Handbook.*

Document Revision History

Table 7–2 shows the revision history for this chapter.

Table 7–2. Documentation Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Updates for the Quartus II software version 6.1.0 release, including: • Moved the "Dual-Clock Synchronous RAM" on page 7–20 section within the chapter • Added information about read-through-write conditions • Added example code, including Examples 7–15 and 7–16; Examples 7–17 and 7–19; and Example 7–23 • Added a section about "Designing with Low-Level Primitives" on page 7–67 • Added information about implementing a safe state machine • Reorganized the chapter, shuffling the "Coding Guidelines for Registers & Latches" and "General Coding Guidelines" and the subsections therein • Added "Comparators" on page 7–65 and "Counters" on page 7–67 to the General Coding Guidelines section	Updates for the Quartus II software version 6.1, including the addition of Stratix III devices. Changes to the recommendations for RAM block inference to ensure better quality of results, and new suggestions for different general logic structures.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	

Table 7–2. Documentation Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
May 2005 v5.0.0	Chapter 4 was formerly Chapter 1 in version 4.2.	
December 2004 v2.1	Updated for Quartus II software version 4.2: • Chapter 4 was formerly Chapter 1. • General formatting and editing updates. • Device family support descriptions updated. • Updated HardCopy structured support for performance improvements. • Quartus II Archive File automatically receives buffer insertion. • Power Calculator now Power Estimator for affected devices. • Updates to tables, figures. • The description of How to Design HardCopy Stratix Devices was updated. • The description of HardCopy Timing Optimization Wizard was updated. • *HardCopy Floorplans & Timing Modules* was renamed to *Design Optimization*. • The description of Performance Estimation was updated. • Added new section on Buffer Insertion. • Location Constraints was updated. • Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices was removed. • A new section *Altera Recommended HDL* was added. • Table 2–5 was added. It lists the HardCopy Stratix design files collected by the hardCopy Files Wizard. • The description of the HardCopy APEX Power Estimator was updated. • A new section about Targeting Designs to HardCopy APEX Devices was added.	

Section III. Synthesis

As programmable logic devices (PLDs) become more complex and require increased performance, advanced design synthesis has become an important part of the design flow. In the Quartus® II software you can use the Analysis & Synthesis module of the Compiler to analyze your design files and create the project database. You can also use other EDA synthesis tools to synthesize your designs, and then generate EDIF netlist files or VQM files that can be used with the Quartus II software. This section explains the options that are available for each of these flows, and how they are supported in the Quartus II, version 6.1 software.

This section includes the following chapters:

- Chapter 8, Quartus II Integrated Synthesis
- Chapter 9, Synplicity Synplify & Synplify Pro Support
- Chapter 11, Mentor Graphics Precision RTL Synthesis Support
- Chapter 10, Mentor Graphics LeonardoSpectrum Support
- Chapter 12, Synopsys Design Compiler FPGA Support
- Chapter 13, Analyzing Designs with Quartus II Netlist Viewers

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

8. Quartus II Integrated Synthesis

QII51008-6.1.0

Introduction

As programmable logic designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. The Quartus® II software includes advanced integrated synthesis that fully supports VHDL and Verilog HDL, as well as Altera®-specific design entry languages, and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use solution.

This chapter documents the design flow and language support in the Quartus II software. It explains how you can use incremental synthesis to reduce your compilation time, and how you can improve synthesis results with Quartus II synthesis options and by controlling the inference of architecture-specific megafunctions. This chapter also explains some of the node-naming conventions used during synthesis to help you better understand your synthesized design, and the messages issued during synthesis to improve your HDL code. Scripting techniques for applying all the options and settings described are also provided.

This chapter contains the following sections:

This chapter provides examples of how to use attributes described within the chapter, but does not cover specific coding examples.

For examples of Verilog HDL and VHDL code synthesized for specific logic functions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For information on 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 process includes Quartus II integrated synthesis, which fully supports the Verilog HDL and VHDL languages as well as Altera-specific languages, and supports a subset of the SystemVerilog language (refer to "Language Support" on page 8–5 for details). This stage of the compilation flow performs logic synthesis to optimize design logic, and performs technology mapping to implement the design logic using device resources such as logic elements (LEs) or adaptive logic modules (ALMs). This stage also generates the single project database that integrates all the design files in a project (including any netlists from third-party synthesis tools).

You can use the Analysis & Synthesis stage of the Quartus II compilation to perform any of the following levels of analysis and synthesis:

- **Analyze Current File**—Parse the current design source file to check for syntax errors. This command does not report on many semantic errors that require further design synthesis. On the Processing menu, click **Analyze Current File**.
- **Analysis & Elaboration**—Check a design for syntax and semantic errors and perform elaboration. On the Processing menu, click **Start**, and then click **Start Analysis & Elaboration**.
- **Analysis & Synthesis**—Perform complete analysis and synthesis on a design, including technology mapping. On the Processing menu, point to **Start**, and 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 using Quartus II integrated synthesis is made up 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. From the File menu, click **New Project Wizard**.
2. Create design files in the Quartus II software or with a text editor.
3. From 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 8–30.

5. Compile the design in the Quartus II software. To synthesize the design, on the Processing menu, point to Start, and click **Start Analysis & Synthesis**.

   On the Processing menu, click **Start Compilation** to run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis.

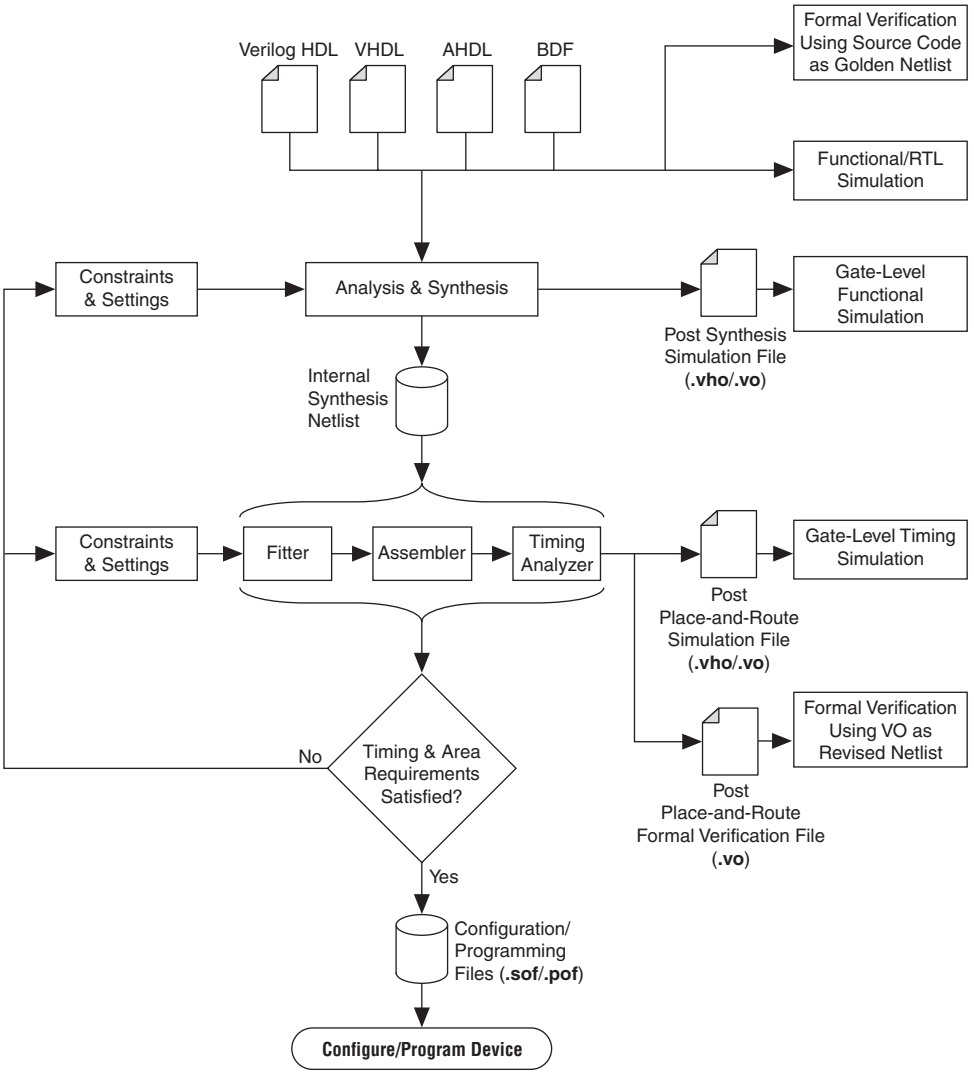
6. After obtaining synthesis and place-and-route results that meet your needs, program the Altera device.

The software provides netlists that allow you to perform functional simulation and gate-level timing simulation in the Quartus II simulator or a third-party simulator, perform timing analysis in a third-party timing analysis tool in addition to the TimeQuest or Classic Timing Analyzer, and/or perform formal verification in a third-party formal verification tool. The Quartus II software also provides many additional analysis and debugging features.

For more information about creating a project, compilation flow, and other features in the Quartus II software, refer to the Quartus II Help. For an overall summary of Quartus II features, refer to the *Introduction to Quartus II Manual*.

Figure 8–1 shows the basic design flow using Quartus II integrated synthesis.

Figure 8–1. Quartus II Design Flow Using Quartus II Integrated Synthesis



Language Support

This section explains the Quartus II software's integrated synthesis support for HDL and schematic design entry, and explains how to specify the Verilog or VHDL language version used in your design. It also documents language features such as Verilog macros, initial constructs and memory system tasks, and VHDL libraries. "Design Libraries" on page 8–13 describes how to compile and reference design units in different custom libraries and ""Using Parameters/Generics" on page 8–18 describes how to use parameters or generics and how to pass them between different languages.

To ensure that the software reads all associated project files, add each file to your Quartus II project. To add files to your project in the Quartus II GUI, on the Project menu, click **Add/Remove Files In Project**. Design files can be added to the project in any order. You can mix all supported languages and netlists generated by third-party synthesis tools in a single Quartus II project.

Verilog HDL Support

The Quartus II Compiler's analysis and synthesis module supports the following Verilog HDL standards:

- Verilog-1995 (IEEE Standard 1364-1995)
- Verilog-2001 (IEEE Standard 1364-2001)
- SystemVerilog-2005 (IEEE Standard 1800-2005) (not all constructs are supported)

For complete information about specific syntax features and language constructs, refer to the Quartus II Help.

The Quartus II Compiler uses the Verilog-2001 standard by default for files that have the extension **.v**.

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified.

To specify a default Verilog HDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the Settings dialog box, under Category, expand **Analysis & Synthesis Settings**, and select **Verilog HDL Input**, then click **OK**.
3. On the **Verilog HDL Input** page, under **Verilog version**, select the appropriate Verilog version, then click **OK**.

You can override the default Verilog HDL version for each Verilog design file by performing the following steps:

1. On the Project File, click **Add/Remove Files in Project**. The **Setting** dialog box appears.
2. On the **Files** page, click on the appropriate file in the list and click the **Properties** button.
3. In the **HDL Version** list, select **SystemVerilog_2005**, **Verilog_2001**, or **Verilog_1995** and click **OK**.

You can also control the Verilog HDL version inside a design file using the `VERILOG_INPUT_VERSION` synthesis directive as shown in Example 8–1. This directive overrides the default HDL version and any HDL Version specified in the File Properties dialog box.

Example 8–1. Controlling the Verilog HDL Input Version with a Synthesis Directive

```
// synthesis VERILOG_INPUT_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`

When the software reads a `VERILOG_INPUT_VERSION` synthesis directive, the current language version changes as specified until the end of the file, or until the next `VERILOG_INPUT_VERSION` directive is reached.

You cannot change the language version in the middle of a Verilog module.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–35.

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 & Setting the HDL Version" on page 8–85.

The Quartus II software support for Verilog HDL is case sensitive in accordance with the Verilog HDL standard. The Quartus II software supports the compiler directive `` `define ``, in accordance with the Verilog HDL standard.

The Quartus II software supports the `include` compiler directive to include files with absolute paths (with either "/" or "\" as the separator), or relative paths (relative to project root, user libraries or current file location). When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the software cannot find the file, it then searches relative to all user libraries, and finally relative to the directory location of the current file.

Verilog-2001 Support

The Quartus II software does not support Verilog-2001 libraries and configurations.

SystemVerilog Support

The Quartus II software supports the following SystemVerilog constructs:

- 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
- Unpacked and packed arrays (does not support packed arrays with more than one dimension)
- 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`.
- Assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`, and `>>>=`
- Increment `++` and decrement `--`
- Jump statements `return`, `break`, and `continue`
- Assignment patterns
- Keywords `unique` and `priority` in case statements
- Default values for function/task arguments

Quartus II integrated synthesis also parses but otherwise ignores SystemVerilog assertions.

Designs written to comply with the Verilog-2001 standard may not compile successfully using the SystemVerilog setting because the SystemVerilog standard adds a number of new reserved keywords. For a list of reserved words in each language standard, refer to the Quartus II Help.

Initial Constructs & Memory System Tasks

The Quartus II software infers power-up conditions from Verilog `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 & On / Synthesis Off & On" on page 8–65. 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.

Quartus II integrated synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories. Example 8–2 shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 8–2. Verilog Example of Initializing RAM with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
  $readmemb("ram.txt", ram);
end
```

When creating a text file to use for memory initialization, specify the address using the format @<*location*> on a new line, then specify the memory word such as `110101` or `abcde` on the next line. Example 8–3 shows a portion of a memory initialization file for the RAM in Example 8–2.

Example 8–3. Text File Format for 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 Macro Default Value in the GUI
To specify a macro in the GUI, on the Assignments menu, click **Settings**. Under **Category**, expand **Analysis & Synthesis Settings** and click **Verilog HDL Input**. Under **Verilog HDL macro**, type the macro name in the **Name** box, the value in the **Setting** box, and click **Add**.

Setting a Verilog Macro Default Value on the Command Line
To set a default value for a Verilog macro on the command-line, use the `--verilog_macro` option as shown in Example 8–4.

Example 8–4. Command Syntax for Specifying a Verilog Macro

```
quartus_map <Design name> --verilog_macro= "<Macro Name>=<Macro Setting>" ↵
```

The command in Example 8–5 has the same effect as specifying `` `define a 2 `` in the Verilog HDL source code:

Example 8–5. Specifying a Verilog 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 8–6:

Example 8–6. Specifying Verilog Macros a = 2 & a = 3

```
quartus_map my_design --verilog_macro="a=2" --verilog_macro="b=3" ↵
```

VHDL Support

The Quartus II Compiler's analysis and synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)

For information about specific syntax features and language constructs, refer to the Quartus II Help.

The Quartus II Compiler uses the VHDL 1993 standard by default for files that have the extension **.vhdl** or **.vhd**.

The VHDL code samples provided in this document follow the VHDL 1993 standard.

To specify a default VHDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the Settings dialog box, under Category, expand **Analysis & Synthesis Settings**, and select **VHDL Input**, then click **OK**.
3. On the **VHDL Input** page, under **VHDL version**, select the appropriate version, then click **OK**.

You can override the default VHDL version for each VHDL design file by performing the following steps:

1. On the Project File, click **Add/Remove Files in Project**. The **Setting** dialog box appears.
2. On the Files page, click on the appropriate file in the list and click **Properties**.
3. In the HDL version list, select **VHDL93** or **VHDL87** and click **OK**.

You can also specify the VHDL version for each design file using the `VHDL_INPUT_VERSION` synthesis directive as shown in Example 8–7. This directive overrides the default HDL version and any HDL Version specified in the **File Properties** dialog box.

Example 8–7. Controlling the VHDL Input Version with a Synthesis Directive

```
--synthesis VHDL_INPUT_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- VHDL87
- VHDL93

When the software reads a VHDL_INPUT_VERSION synthesis directive, it changes the current language version as specified until the end of the file, or until it reaches the next VHDL_INPUT_VERSION directive.

You cannot change the language version in the middle of a VHDL design unit.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–35.

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 & Setting the HDL Version" on page 8–85.

The Quartus II software reads default values for registered signals defined in the VHDL code and converts the default values into power-up level settings. This enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.

VHDL Standard Libraries & Packages

The Quartus II software includes the standard IEEE libraries and a number of vendor-specific VHDL libraries. For information about organizing your own design units into custom libraries, refer to "Design Libraries" on page 8–13.

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 packages standard (included in every project by default) and textio. For compatibility with older designs, the Quartus II software also supports the following vendor-specific packages and libraries:

- Synopsys packages such as std_logic_arith and std_logic_unsigned in the **IEEE** library

- Mentor Graphics® packages such as `std_logic_arith` in the **ARITHMETIC** library
- Altera primitive packages `altera_primitives_components` (for primitives such as `GLOBAL` and `DFFE`) and `maxplus2` (for legacy support of MAX+PLUS® II primitives) in the **ALTERA** library
- Altera megafunction packages `altera_mf_components` and `stratixgx_mf_components` in the **ALTERA_MF** library (for Altera-specific megafunctions including LCELL), and `lpm_components` in the **LPM** library for library of parameterized modules (LPM) functions.

For a complete listing of library and package support, refer to the Quartus II Help.

Beginning with the Quartus II software version 5.1, you should import component declarations for Altera primitives such as `GLOBAL` and `DFFE` from the `altera_primitives_components` package and not the `altera_mf_components` package.

AHDL Support

The Quartus II Compiler's analysis and synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (**.tdf**). You can import AHDL Include Files (**.inc**) into a Text Design File with an AHDL `include` statement. Altera provides AHDL Include Files for all megafunctions shipped with the Quartus II software.

For information about specific syntax features and language constructs, refer to the Quartus II Help.

The AHDL language does not support the synthesis directives or attributes described in this chapter.

Schematic Design Entry Support

The Quartus II Compiler's analysis and synthesis module fully supports Block Design Files (**.bdf**) for schematic design entry.

You can use the Quartus II software's Block Editor to create and edit Block Design Files and open Graphic Design Files (**.gdf**) imported from the MAX+PLUS II software. Use the Symbol Editor to create and edit Block Symbol Files (**.bsf**) and open MAX+PLUS II Symbol Files (**.sym**). You can

read and edit these legacy MAX+PLUS II formats with the Quartus II Block and Symbol Editors; however, the Quartus II software saves them as **.bdf** or **.bsf** files.

For information about creating and editing schematic designs, refer to the Quartus II Help.

Schematic entry methods do not support the synthesis directives or attributes described in this chapter.

Design Libraries

By default, the Quartus II software compiles all design files into the work library. If you do not specify a design library, or if a file refers to a library that does not exist, or if the library does not contain a referenced design unit, then the software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup. (Creating separate custom design libraries is optional.)

To compile your design files into specific libraries, (for example, when you have two or more functionally different design entities that share the same name) you can specify a destination library for each design file in various ways, as described in the following subsections:

- "Specifying a Destination Library Name in the Settings Dialog Box" on page 8–14
- "Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl" on page 8–14

When the Quartus II Compiler analyzes the file, it stores the analyzed design units in the file's destination library.

Beginning with the Quartus II software version 6.1, a design can contain two or more entities with the same name if they are compiled into separate libraries.

When compiling a design instance, the Quartus II software initially searches for the entity in the library associated with the instance (which is the work library if no other library is specified). If the entity definition is not found, the software searches for a unique entity definition in all design libraries. If more that 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 8–15. In Verilog HDL, BDF, 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:

1. From the Assignments menu, choose **Settings**.
2. On the **Files** page of the **Settings** dialog box, specify the library name in the **File Name** list.
3. Click **Properties**.
4. In the **File Properties** dialog box, select the type of design file from the **Type** list.
5. Type the desired library name in the **Library** field.
6. 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 Setting File or with Tcl commands.

For example, the following Quartus II Settings File or Tcl assignments specify that the Quartus II software analyzes **my_file.vhd** and store its contents (design units) in the VHDL library **my_lib**, and analyzes the Verilog file **my_header_file.h** and store its contents in a library called **another_lib**.

Example 8–8. 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 8–85.

Specifying a Destination Library Name in a VHDL File

You can use the `library` synthesis directive to specify a library name in your VHDL source file. This directive takes a single string argument: the name of the destination library. Specify the `library` directive in a VHDL comment prior to the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), using one of the supported keywords for synthesis directives, that is, `altera`, `synthesis`, `pragma`, `synopsys`, or `exemplar`.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–35.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file through the **Settings** dialog box, any existing Quartus II Settings File setting, any setting made through the Tcl interface, or any prior `library` directive in the current file. The directive remains effective until the end of the file or the next `library` synthesis directive.

Example 8–9 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 8–9. 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 the **Settings** dialog box, editing the Quartus II Settings File, or using the Tcl interface. Using the `library` directive to change the destination VHDL library within a source file gives you the option to organizing the design units in a single file into different libraries, rather than just a single library.

The Quartus II software produces an error if you use the library directive within a design unit.

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides a number of ways to map or bind an instance to an entity in a specific library, as described in the following subsections.

Direct Entity Instantiation

In the direct entity instantiation method, the instantiation refers to an entity in a specific library, as shown in Example 8–10.

Example 8–10. VHDL Example of Direct Entity Instantiation

```
entity lib1.foo is
port( ... );
end entity lib1.foo;

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 Indication

There is more than one mechanism for binding a component to an entity. In an explicit binding indication, you provide an explicit binding indication for a particular component instantiation, as shown in Example 8–11.

Example 8–11. VHDL Example of Explicit Binding Instantiation

```
entity entity1 is
port(...);
end entity entity1;

package entities is
   component entity1 is
      port map (...);
   end component entity1;
end package entities;

entity top_entity is
port(...);
end entity top_entity;

use lib1.entities.all;
architecture arch of top_entity is
for I1: entity1 use
   entity lib1.entity1
     port map(...);
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 entity with the same name. If no such entity is visible in the current scope, then the instance is bound to the library in which the component was declared. For example, if the component is declared in a package in library `MY_LIB`, then an instance of the component would be bound to the entity in library `MY_LIB`. The portions of code in Example 8–12 show this instantiation method.

Example 8–12. VHDL Example of Default Binding Instantiation

```
use lib1.foo;
entity entity1 is
port(...);
end entity entity1;

architecture arch of entity1 is
begin
-- instance inst uses the explicit visible
-- declaration of foo from lib1
   inst: foo
   port map(...);
end architecture arch;
```

Using Parameters/Generics

This section describes how parameters (called generics in VHDL) are supported in the Quartus II software, and how you can pass these parameters between different design languages.

You can enter default parameter values for your design in the **Default Parameters** box in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. You can also specify parameters for instantiated modules in a Block Design File. To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab. For these GUI-based entry methods, refer to "Setting Default Parameter Values & BDF Instance Parameter Values" for information about how parameter values are interpreted, and for recommendations about the format you should use.

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 may instantiate a VHDL entity from a Verilog design file. You can pass parameters or generics between VHDL, Verilog HDL, AHDL, and BDF schematic entry, and from EDIF or VQM to any of these languages. In most cases, you do not have to do anything special to pass parameters from one language to another. However, in some cases you may have to specify the type of the parameter you are passing. In those cases you should follow certain guidelines to ensure that the parameter value is interpreted correctly. Refer to "Passing Parameters Between Two Design Languages" on page 8–19 for parameter type rules.

Setting Default Parameter Values & 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, you can use `'1` as the parameter value, and to pass a 4-bit binary vector from BDF to Verilog, you can use `4'b1111` for a parameter of a Verilog entity.

In a few cases, the Quartus II software cannot infer the correct type of a parameter value. To avoid ambiguity, specify the parameter value in a type-encoded format where the first or first and second character of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string `1010` from BDF to Verilog HDL, you cannot simply use the value `1001`, because the Quartus II software interprets it as a decimal value. You also can not use the string `"1001"`, because the Quartus II software interprets it as an ASCII string. You must use the type-encoded string `B"1001"` for the Quartus II software to interpret the parameter value correctly. Table 8–1 provides a list of valid parameter strings and how they are interpreted within the Quartus II software. Altera recommends using the type-encoded format only when necessary to resolve ambiguity.

Table 8–1. Valid Parameter Strings & How They are Interpreted

Parameter String	Quartus II Parameter Type, Format & Value
S"abc", s"abc"	String value “abc”
"abc123", "123abc"	*String value abc123 or 123abc*
F"12.3", f"12.3"	Floating point number 12.3
-5.4	*Floating point number -5.4*
D"123", d"123"	Decimal number 123
123, -123	*Decimal number 123, -123*
X"ff", H"ff"	Hexadecimal value FF
Q"77", O"77"	Octal value 77
B"1010", b"1010"	Unsigned binary value 1010
SB"1010", sb"1010"	Signed binary value 1010
R"1", R"0", R"X", R"Z", r"1", r"0", r"X", r"Z"	Unsized bit literal
E"apple", e"apple"	Enum type, value name is apple
P"1 unit"	Physical literal, the value is (1, unit)
A(…), a(…)	Array type or record type, whose content is determined by the string (...)

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 the parameter information. It is essential that the parameter be correctly interpreted by the subdesign language (the design entity that is instantiated). Based on the information provided

by the higher-level design and the value format, and sometimes by the parameter type of the subdesign entity, the Quartus II software interprets the type and value of the passed parameter.

When passing a parameter whose value is an enumerated type value or literal from a language that does not support enumerated types to one that does (for example from Verilog to VHDL), it is essential that the enumeration literal is spelled correctly in the higher-level design. The parameter value is passed as a string literal, and it is up to the language of the lower-level design to correctly convert the string literal into the correct enumeration literal.

If the lower-level language is SystemVerilog, it is essential that the enum value is used in the correct case. In SystemVerilog, it is recommended that two enumeration literals do not only differ in case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion among users and it is more difficult to pass parameters from case-insensitive HDLs, such as VHDL. Arrays have different support in different design languages. For details about the array parameter format, refer to the **Parameter** section in the Analysis & Synthesis report of a design that contains array parameters or generics.

The following code show examples of passing parameters from one design entry language to a subdesign written in another language. Example 8–13 shows a VHDL subdesign that is instantiated into a top-level Verilog design in Example 8–14. Example 8–15 shows a Verilog subdesign that is instantiated in a top-level VHDL design in Example 8–16.

Example 8–13. 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 8–14. Verilog HDL Top-level Design Instantiating and Passing Parameters to VHDL Entity from Example 8–13

```
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 8–15. Verilog HDL Parameterized Subdesign Module

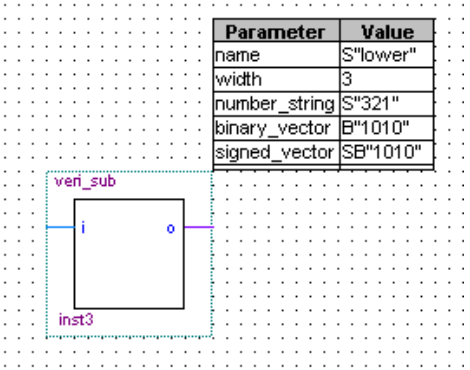
```
module veri_sub (...)
parameter name = "default";
parameter width = 8;
parameter number_string = "123";
parameter binary_vector = 4'b0101;
parameter signed_vector = 4'sb1111;
```

Example 8–16. VHDL Top-level Design Instantiating and Passing Parameters to the Verilog Module from Example 8–15

```
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 8–15 in a top-level BDF design, you must first generate a symbol for the HDL file, as shown in Figure 8–2. Open the HDL file in the Quartus II software, then from the File menu, point to Create/Update and click **Create Symbol Files for Current File**. To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab.

Figure 8–2. BDF Top-level Design Instantiating and Passing Parameters to the Verilog Module from Example 8–15.



Incremental Synthesis

The incremental synthesis feature in the Quartus II software manages a design hierarchy for incremental design by allowing you to divide the design into multiple partitions. Incremental synthesis ensures that when a design is compiled, only those partitions of the design that have been updated will be resynthesized, reducing synthesis time and runtime memory usage. You can change and resynthesize a design partition without affecting other design partitions, which means that node names are maintained during synthesis for all registered and combinational nodes in unchanged partitions.

Conventionally, a hierarchical design is flattened into a single netlist of logic gates before logic synthesis and technology mapping. However, incremental synthesis allows you to partition a hierarchical design along any of its hierarchical boundaries. The individual hierarchical partitions are synthesized and mapped separately by the Quartus II software. The hierarchical partitions are then combined—or merged—to form a flattened netlist for further stages of the Quartus II compilation flow, including fitting. The mapped netlist for each partition is stored by the Quartus II software. Therefore, if the source code for one partition changes during the design cycle, only the partition that changed is resynthesized during the next compilation of the design.

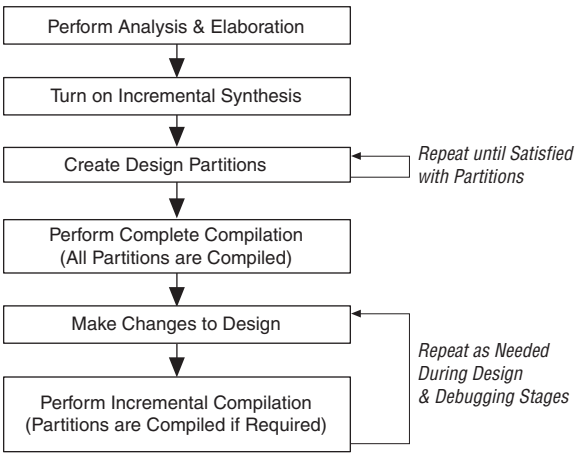
You can use incremental synthesis by itself, or use a full incremental compilation flow in which you also preserve the placement (and optionally routing) information for unchanged partitions. Some Quartus II features such as formal verification and incremental SignalTap® II logic analysis require that full incremental compilation is turned on. If your design flow requires the full incremental compilation

feature, you can perform incremental synthesis by using full incremental compilation with the Netlist Type for all design partitions set to Post-Synthesis.

This chapter describes incremental synthesis only. For information about the full incremental compilation flow, and the differences between full incremental compilation and incremental synthesis, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The flow chart in Figure 8–3 shows the steps in the incremental synthesis flow.

Figure 8–3. Summary of Design Flow Using Quartus II Incremental Synthesis



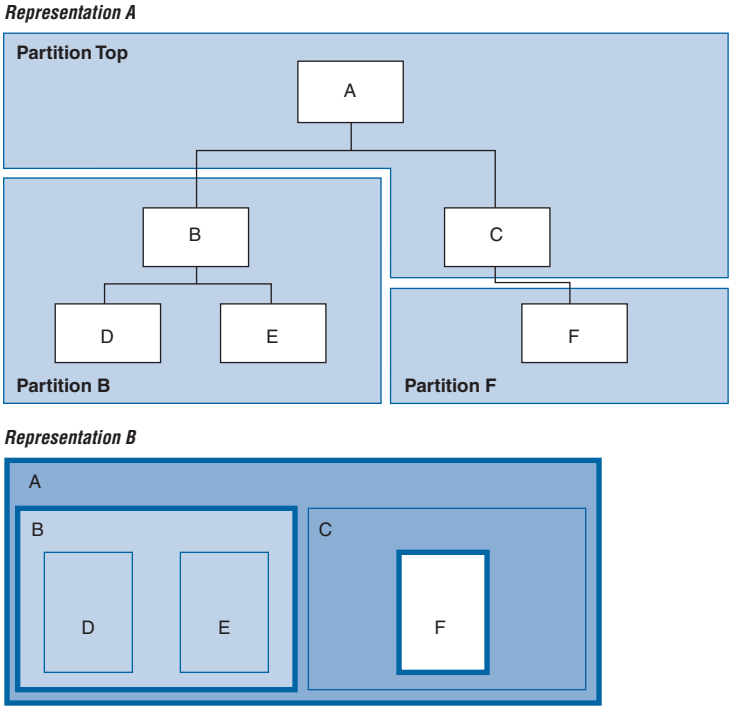
Partitions for Incremental Synthesis

A partition represents a portion of the design that you want to synthesize incrementally. Partitions must be bounded by hierarchical boundaries, and therefore, cannot be a portion of the logic within a hierarchical block. When a partition is declared, every hierarchical block within that partition becomes part of the same partition. You can create new partitions for hierarchical blocks within an existing partition, in which case the blocks designates as a new partition are no longer part of the higher-level partition.

In Figure 8–4, hierarchical entities B and F form partitions in the complete design, which is made up of entities A, B, C, D, E, and F. The shaded boxes in Representation A indicate design partitions in a "tree" representation

of the hierarchy. In Representation B, the lower-level entities are represented inside of higher-level entities, and the partitions are illustrated with different colored shading. The top-level partition Top automatically contains the top-level entity in the design, and any logic that is not defined as part of another partition. The design file for the top-level may be just a wrapper for the hierarchical entities below it, or it may contain its own logic. In this example, the partition for top-level entity A also includes the logic in one of its lower-level entities, C. Because entity F is contained in its own partition, it is not treated as part of the top-level partition. Another separate partition, B, contains the logic in entities B, D, and E.

Figure 8–4. Partitions in a Hierarchical Design



Partitions for Preserving Hierarchical Boundaries

Use partitions if you need to preserve hierarchical boundaries through the synthesis process. For example, if you are performing formal verification, you must use partitions with the full incremental compilation flow to ensure that no optimizations occur across specific design hierarchies.

Follow the steps described in "Preparing a Design for Incremental Synthesis" on page 8–25 if you need to set up your design to preserve hierarchical boundaries during Quartus II synthesis. If required, for example when using incremental SignalTap II logic analysis, you can use partitions with full incremental compilation (instead of incremental synthesis only) to preserve boundaries throughout the entire compilation process.

Beginning with the Quartus II software version 6.0, Altera recommends that you use Design Partition assignments instead of the Preserve Hierarchical Boundary logic option, which may be removed in future versions of the Quartus II software.

Preparing a Design for Incremental Synthesis

To set up your design with partitions for incremental synthesis, identify the design partitions and turn on incremental synthesis using the following steps:

1. On the Processing menu, point to Start and click **Start Analysis & Elaboration** to elaborate the design, or perform any compilation flow that includes this step. This allows the Quartus II software to identify your design's hierarchy.

2. Identify the partitions in your design by applying the `PARTITION_HIERARCHY` assignment to the appropriate instances. You can do this using the list of instances under **Compilation Hierarchy** in the **Project Navigator**. Right-click on an instance in the **Project Navigator** and click **Set as Design Partition**.

   An incremental compilation icon appears next to each instance that is set as a partition.

3. On Assignments menu, click **Settings**. The **Settings** dialog box appears.

4. On the **Compilation Process Settings** page of the **Settings** dialog box, select **Incremental synthesis only** in the **Incremental compilation** section.

Full incremental compilation is the default setting beginning in the Quartus II software version 6.1.

To remove an existing PARTITION_HIERARCHY assignment with the GUI, right-click the instance in the **Project Navigator** and select **Set as Design Partition** to turn off the option.

Synthesizing a Design Using Incremental Synthesis

Once incremental synthesis is enabled, it becomes part of the compilation procedure under normal circumstances, that is, when you click **Start Compilation** from the Processing menu, or when you select **Start Compilation** in the toolbar.

During compilation, the software synthesizes each partition separately, and then merges the partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting.

For subsequent iterations of analysis and synthesis, the Quartus II software uses an internal checksum calculation to determine whether the file should be resynthesized. The software checks the contents of the source file, as well as the values of the synthesis and netlist optimization settings.

When you are using the full incremental compilation flow, changes in settings do not trigger automatic resynthesis of the design when you specify that you want to use any existing netlist. For more information about the differences between full incremental compilation and incremental synthesis, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The Quartus II software resynthesizes only those partitions that contain changed source code, or changes to synthesis or netlist optimization settings. If you modify a file in the top-level partition, but none of the lower-level partitions are affected, the Quartus II software only resynthesizes the top-level partition.

The software always maintains the synthesis results for unchanged partitions, and merges newly synthesized partitions with unchanged partitions.

Synthesizing Using the Synthesis & Merge Commands

If you compile your design using the individual compilation steps available from the Start submenu of the Processing menu or by selecting **Compilation Tool** from the Tools menu, use the separate synthesis and merge commands.

Once incremental synthesis is enabled, on the Processing menu, point to **Start** and click **Start Analysis & Synthesis** to separately synthesize each partition. You must then merge the partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting. On the Processing menu, point to **Start** and click **Start Partition Merge**. After each incremental iteration of analysis and synthesis, merge the newly synthesized partitions with the unchanged partitions.

Forcing Complete Resynthesis

Because the incremental synthesis flow identifies changes to source code and assignments and resynthesizes only the partitions that have changed, you usually do not have to completely resynthesize all of your source files.

If you want to force complete resynthesis, from the **Incremental compilation** section of the **Compilation Process** page of the **Settings** dialog box, select **Off**. Then resynthesize your entire design, and turn on **Incremental synthesis only** again (if desired). Alternately, to save the extra synthesis run, you can make a small change and save at least one file in each design partition, so that the Quartus II software detects the changes and resynthesizes each partition on the next analysis and synthesis.

Considerations & Restrictions When Using Incremental Synthesis

To use incremental synthesis effectively, there are some issues to consider when planning your design's structure. Additionally, there are restrictions when using incremental synthesis with other Quartus II features. This section provides information about the following considerations and restrictions:

- Hierarchical considerations
- Restrictions on megafunction partitions
- Resource balancing
- Back-annotating node locations using the Altera LogicLock™ design methodology
- SignalTap® II logic analyzer

Hierarchical Considerations

When planning a design, keep in mind the size and scope of each partition, and how likely it is that different parts of your design will change as your design develops.

For guidelines about design hierarchical partitioning, refer to the *Hierarchical Design Partitioning* section of the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

Observe the following important hierarchical design considerations:

- Register all inputs and outputs of each block. This helps avoid any delay penalty on signals that cross partition boundaries.

  While this can be difficult in practice, greater adherence to this principle results in less timing degradation and area increase when using incremental flows. Registering lessens the need for the cross-partition optimizations that are prevented by partitioning.

- Do not use tri-state signals or bidirectional ports on hierarchical boundaries unless they are directly connected to top-level pins. If you use boundary tri-states in a lower-level block, synthesis pushes the tri-states through the hierarchy to the top-level to take advantage of the tri-state drivers on the output pins of the device. Because this requires optimizing through hierarchies, lower-level boundary tri-state signals are restricted with a block-level or incremental design methodology.

  Using incremental synthesis, internal tri-states are supported only when all the destination logic is contained in the same partition, in which case analysis and synthesis implements the internal tri-state signals using multiplexing logic. For bidirectional ports that feed a bidirectional pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition.

- Remember that logic is not synthesized, or optimized, across partition boundaries, which means any constant value (e.g., signals set to GND) will not be propagated across partitions.

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 the creation of a partition for inferred megafunctions (that is, in which the software uses 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 creation of a partition for any Quartus II internal hierarchy that is dynamically-generated during compilation to implement the contents of a megafunction.

Resource Balancing

You may have to do some manual resource balancing across partitions. When using incremental synthesis, each partition is synthesized separately, with no data about the resources used in other partitions. This means device resources could be overused in the individual partitions during synthesis, and thus the design may not fit in the target device when the partitions are merged.

For example, in the regular synthesis flow, when DSP blocks or RAM blocks are overused, the Quartus II Compiler can perform resource balancing and convert some of the logic into regular logic cells, for example, LEs or adaptive logic modules (ALMs). Without data about resources used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource such that the design does not fit once all the partitions are merged. In this case, you may be able to manually balance the resources by using the Quartus II synthesis options to control inference of megafunctions that use the DSP or RAM blocks.

Refer to "Megafunction Inference Control" on page 8–53 for more information about resource balancing. 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.

For more tips on resource balancing and reducing resource utilization, refer to the appropriate section of the *Area & Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Preserving Compilation Results

Altera recommends using full incremental compilation for top-down and bottom-up compilation flows where you wish to preserve placement and routing information and the performance of unchanged parts of your design. The full incremental compilation feature also allows you to export and import lower-level design files to enable team-based design flows or design flows where you need to optimize different blocks separately.

For more information about preserving results and exporting or importing design blocks using full incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

OpenCore Plus MegaCore Functions

The circuitry involved in providing OpenCore® Plus MegaCore® functions is currently incompatible with user-created design partitions

Quartus II Synthesis Options

The Quartus II software offers a number of options to help you control the synthesis process and achieve the optimal results for your design. The "Setting Synthesis Options" on page 8–32 section describes the synthesis settings dialog box where you can set the most common global settings and options, and defines the following three types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives. The other subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples of how to use each option where applicable:

- Major Optimization Settings
  - "Optimization Technique" on page 8–36
  - "Speed Optimization Technique for Clock Domains" on page 8–36
  - "PowerPlay Power Optimization" on page 8–37
  - "Refer to "Partitions for Preserving Hierarchical Boundaries" on page 8–25 for details about using design partitions." on page 8–47
- State Machine Settings and Enumerated Types
  - "State Machine Processing" on page 8–40
  - "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 8–41
  - "Manually Specifying Enumerated Types Using the enum_encoding Attribute" on page 8–42
  - "Safe State Machines" on page 8–43
- Register Power-Up Settings

Setting Synthesis Options

You can set synthesis options in the **Settings** dialog box, or with logic options in the Quartus II software, or you can use synthesis attributes and directives within the HDL source code.

Analysis & Synthesis Page of the Settings Dialog Box

On the Assignments menu, click **Settings** to open the **Settings** dialog box. The **Analysis & Synthesis Settings** page allows you to set global synthesis options that apply to the entire project. These options are described in later subsections.

Quartus II Logic Options

Quartus II logic options control many aspects of the synthesis and place-and-route process. To set logic options in the Quartus II graphical user interface, on the Tools menu, click **Assignment Editor**. You can also use a corresponding Tcl command. Quartus II logic options allow you to set instance or node-specific assignments without editing the source HDL code. Logic options can be used with all design entry languages supported by the Quartus II software.

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 user interface or with Tcl. Each attribute description in this chapter indicates whether there is a corresponding setting or logic option that can be set in the user interface; some attributes can be specified only with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the user interface or in the Quartus II Settings File. Assignments or settings made through the Quartus II user interface, the Quartus II Settings File, or the Tcl interface take precedence over assignments or settings made with synthesis attributes in your HDL code.

The Verilog-2001, SystemVerilog, and VHDL language definitions provide specific syntax for specifying attributes, but in Verilog-1995 HDL, you must embed attribute assignments in comments. You can enter attributes in your code using the syntax in Examples 8–17, 8–18, and 8–19, where <*attribute*>, <*attribute type*>, <*value*>, <*object*>, and <*object type*> are variables, and the entry in brackets is optional. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case-sensitive, therefore, synthesis attributes are also case sensitive.

Example 8–17. Synthesis Attributes in Verilog-1995 HDL

```
// synthesis <attribute> [ = <value> ]
                    or
/* synthesis <attribute> [ = <value> ] */
```

Verilog-1995 comment-embedded attributes as shown in Example 8–17 must be used as a suffix to (that is, placed after) the declaration of an item and must appear before the semicolon when one is required.

You cannot use the open one-line comment in Verilog HDL when a semicolon is required at the end of the line because it is not clear to which HDL element the attribute applies. For example, you cannot make an attribute assignment such as `reg r; // synthesis` <*attribute*> because the attribute could be read as part of the next line.

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces, as follows:

`//synthesis` <*attribute1*> [= <*value*>] <*attribute2*> [= <*value*>]

For example, to set the `maxfan` attribute to `16` (Refer to "Maximum Fan-Out" on page 8–51 for details) and set the `preserve` attribute (Refer to "Preserve Registers" on page 8–48 for details) on a register called `my_reg`, use the following syntax:

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

In addition to the `synthesis` keyword as 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 8–18. Synthesis Attributes in Verilog-2001 & SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Verilog-2001 attributes as shown in Example 8–18 must be used as a prefix to (that is, placed before) a declaration, module item, statement, or port connection, and used as a suffix to (that is, placed after) an operator or a Verilog HDL function name in an expression.

Because formal verification tools do not recognize the syntax, the Verilog-2001 attribute syntax is not supported when using formal verification.

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas, as shown in the following example:

(* <*attribute1*> [= <*value1*>], <*attribute2*> [= <*value2*>] *)

For example, to set the `maxfan` attribute to `16` (Refer to"Maximum Fan-Out" on page 8–51 for details) and set the `preserve` attribute (Refer to "Preserve Registers" on page 8–48 for details) on a register called `my_reg`, use the following syntax:

```
(* preserve, maxfan = 16 *) reg my_reg;
```

Example 8–19. Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

VHDL attributes, as shown in Example 8–19, declare the attribute type and then apply it to a specific object. For VHDL designs, all supported synthesis attributes are declared in the `altera_syn_attributes` package in the Altera library. You can call this library from your VHDL code to declare the synthesis attributes, as follows:

```
LIBRARY altera;
USE altera.altera_syn_attributes.all;
```

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called compiler directives or pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not standard Verilog HDL or VHDL commands; synthesis tools use directives to control the synthesis process in a particular manner. Directives to 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 following syntax shown in Examples 8–20 and 8–21, where <*directive*> and <*value*> are variables, and the entry in brackets is optional. Notice that for synthesis directives there is no = sign before the value; this is different than the syntax for synthesis attributes. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case-sensitive, therefore, all synthesis directives are also case sensitive.

Example 8–20. Synthesis Directives in Verilog HDL

```
// synthesis <directive> [ <value> ]
                    or
/* synthesis <directive> [ <value> ] */
```

Example 8–21. Synthesis Directives in VHDL

```
-- synthesis <directive> [ <value> ]
```

In addition to the `synthesis` keyword shown above, the `pragma`, `synopsys`, and `exemplar` keywords are supported in both Verilog HDL and VHDL for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis directives that will be recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis directive.

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 8–2 lists the settings for this logic option, which you can apply only to a design entity. You can also set this logic option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box.

Table 8–2. Optimization Technique Settings

Setting	Description
Area	The Compiler makes the design as small as possible to minimize resource usage
Speed	The Compiler chooses a design implementation that has the fastest f_{MAX}
Balanced *(1)*	The Compiler maps part of the design for area and part for speed, providing better area utilization than optimizing for speed, with only a slightly slower f_{MAX} than optimizing for speed

Note to Table 8–2:
(1) Balanced optimization technique is not supported for all device families.

The default setting varies by device family, and is generally optimized for the best area/speed trade-off. Results are design-dependent and vary depending on which device family you use.

Speed Optimization Technique for Clock Domains

The **Speed Optimization Technique for Clock Domains** logic option specifies that all combinational logic in or between the specified clock domain(s) is optimized for speed.

When this option is set on a particular clock signal, all the logic in this clock domain is optimized for speed during synthesis. The remainder of the design in other clock domains is synthesized with the project-wide **Optimization Technique** that is set in the **Analysis & Synthesis Settings**. The option can also be set from one clock to another clock signal, in which case the logic in paths from registers the first clock domain to registers in the second clock domain are synthesized for speed. The advantage of using this option over the project-wide setting to optimize for speed is that there is less penalty to the area of the design, because a smaller part of the circuit is optimized for speed. This may also have a positive effect on the clock speed. This option also has an advantage over setting the **Optimization Technique** on a design entity, because that option forces the hierarchical blocks to be synthesized separately. Doing so may increase area and decrease performance due to the lack of optimizations across hierarchies. The **Speed Optimization Technique for Clock**

Domains option does not treat hierarchical entities separately, and can optimize across hierarchical boundaries for logic within the same clock domain.

This option is useful if you have one or more clock domains that do not meet your timing requirements. When there are failing paths within a clock domain, the option can be set on the clock of that clock domain. When there are failing paths between clock domains, the option can be set from one clock domain to the other one.

This option is available for the following device families:

- Stratix® series
- Cyclone™ series
- HardCopy® II
- HardCopy Stratix
- MAX® II

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. On the Assignments menu, click **Settings**, under **Category**, click **Analysis & Synthesis Settings**, this displays the **Analysis & Synthesis Settings** page. The following three settings are available for the PowerPlay power optimization option:

- **Off**—Analysis & Synthesis does not perform any power optimizations.
- **Normal Compilation**—Analysis & Synthesis performs power optimizations, without reducing design performance.
- **Extra Effort**—Analysis & Synthesis performs additional power optimizations which may reduce design performance.

For more information about optimizing your design for power utilization, refer to the *Power Optimization* chapter in volume 3 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*.

Restructure Multiplexers

This option specifies whether the Quartus II software should extract and optimize buses of multiplexers during synthesis.

This option is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of LEs or ALMs. This option is available for the Stratix series, Cyclone series, and MAX II 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 vectors in Verilog HDL, or `STD_LOGIC_VECTOR` signals in VHDL. The **Restructure Multiplexers** option identifies buses of multiplexer trees that have a similar structure. When it is turned on, the **Restructure Multiplexers** option optimizes the structure of each multiplexer bus for the target device to reduce the overall amount of logic used in the design.

Results of the multiplexer optimizations are design dependent, but area reductions as high as 20% are possible. The option may negatively affect your design's f_{MAX}.

Table 8–3 lists the settings for the logic option, which you can apply only to a design entity. You can also specify this option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box for your whole project.

Table 8–3. Restructure Multiplexers Settings

Setting	Description
On	Enables multiplexer restructuring to minimize your design area. This setting may reduce the f_{MAX}.
Off	Disables multiplexer restructuring to avoid possible reductions in f_{MAX}.
Auto (Default)	Allows the Compiler to determine whether to enable the option based on your other Quartus II synthesis settings. The option is On when the **Optimization Technique** option is set to **Area** or **Balanced**, and Off when the **Optimization Technique** option is **Speed**. (Note that since the default Optimization Technique is Balanced for many device families including Stratix series, this option is turned on by default for those families.)

After you have compiled your design, you can view multiplexer restructuring information in the **Multiplexer Restructuring Statistics** report in the **Multiplexer Statistics** folder under **Analysis & Synthesis Optimization Results** in the **Analysis & Synthesis** section of the **Compilation Report**. Table 8–4 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 8–4. Multiplexer Information in the Multiplexer Restructuring Statistics Report

Heading	Description
Multiplexer Inputs	The number of different choices that are multiplexed together.
Bus Width	The width of the bus in bits.
Baseline Area	An estimate of how many logic cells are needed 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 needed to implement the bus of multiplexers if **Multiplexer Restructuring** is applied.
Saving if Restructured	An estimate of how many logic cells are saved if **Multiplexer Restructuring** is applied.
Registered	An indication of whether registers are present on the multiplexer outputs. **Multiplexer Restructuring** uses the secondary control signals of a register (such as synchronous clear and synchronous-load) to further reduce the amount of logic needed to implement the bus of multiplexers.
Example Multiplexer Output	The name of one of the multiplexers' outputs. This name can help determine where in the design the multiplexer bus originated.

For more information about optimizing for multiplexers, refer to the *Multiplexers* section of the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

State Machine Processing

This logic option specifies the processing style used to compile a state machine. Table 8–5 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 8–5. State Machine Processing Settings

Setting	Description
Auto (Default)	Allows the Compiler to choose what it determines to be the best encoding for the state machine
Minimal Bits	Uses the least number of bits to encode the state machine
One-Hot	Encodes the state machine in the one-hot style
User-Encoded	Encodes the state machine in the manner specified by the user

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*.

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. For example, in a one-hot-encoded state machine with five states including an initial or reset state, the software uses the following register encoding:

```
State 0     0 0 0 0 0
State 1     0 0 0 1 1
State 2     0 0 1 0 1
State 3     0 1 0 0 1
State 4     1 0 0 0 1
```

If the State Machine Processing logic option is set to **User-Encoded** in a Verilog HDL design, then 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:

```
parameter S0 = 4'b1010, S1 = 4'b0101, ...
```

If the software infers states `S0, S1, . . .` it uses the encoding `4'b1010, 4'b0101, . . .`. If necessary, the software inverts bits in a user-encoded state machine to ensure that all bits of the reset state of the state machine are zero.

To assign your own state encoding with the **User-Encoded** State Machine Processing option in a VHDL design, you must apply specific binary encoding to the elements of an enumerated type because enumeration literals have no numeric values in VHDL. Use the `syn_encoding` synthesis attribute to apply your encoding values. Refer to "Manually Specifying State Assignments Using the syn_encoding Attribute".

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 8–40. 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, you must use the `syn_encoding` synthesis attribute to apply specific binary encodings to the elements of an enumerated type.

In Example 8–22, 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 8–22. Example of the syn_encoding VHDL Attribute

```
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
```

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 wish to control the encoding for a recognized state machine, use the State Machine Processing logic option and the `syn_encoding synthesis` attribute.

To use the `enum_encoding` attribute in a VHDL design file, associate the attribute with the Enumeration Type whose encoding you want to control. The `enum_encoding` attribute must follow the Enumeration Type Definition but precede its use. In addition, the attribute value must be a string literal that specifies either an arbitrary user encoding or an encoding style of `"default"`, `"sequential"`, `"gray"`, or `"one-hot"`.

An arbitrary user encoding consists of a space-delimited list of encodings. The list must contain as many encodings as there are enumeration literals in your Enumeration Type. In addition, the encodings must all have the same length, and each encoding must consist solely of values from the `std_ulogic` type declared by the `std_logic_1164` package in the `IEEE` library. In the code fragment of Example 8–23, the `enum_encoding` attribute specifies an arbitrary user encoding for the Enumeration Type `fruit`.

Example 8–23. Specifying an Arbitrary User Encoding for Enumerated Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "11 01 10 00";
```

In this example, the enumeration literals are encoded as:

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

Sometimes you may wish to specify an encoding style, rather than a manual user encoding, especially when the Enumeration Type has a large number of enumeration literals. The Quartus II software can implement Enumeration Types with four different encoding styles:

- "default"—Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than 5 literals, use the "sequential" encoding. If there are more than five but fewer than 50 literals, use a "one-hot" encoding. Otherwise, use a "gray" encoding.
- "sequential"—Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding 0, the second 1, and so on.
- "gray"—Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit.
- "one-hot"—The default encoding style requiring *N* bits, where *N* is the number of enumeration literals in the Enumeration Type.

Observe that in Example 8–23, 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 8–24.

Example 8–24. 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 safe attribute 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 encoding register values do not correspond to any of the defined states. 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. In some designs that are operated at high temperatures, designers are also concerned that alpha particles could change the state of the circuit. By default, the behavior of the state machine that enters an illegal state is undefined. However, you can set the safe attribute or logic option on a state machine if you want the state machine to recover deterministically from an illegal state.

It is important to note that the `safe state machine` setting 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 minimizes or removes the logic. Any flag signals or logic used in the design to indicate such an illegal state are also removed. If the state machine is implemented as safe, the recovery logic forces its transition from an illegal state to the reset state.

The **Safe State Machine** option can be set globally, or on individual state machines. To set this option, perform the following steps:

1. From the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. In the **Existing option settings** list, select **Safe State Machine**.
5. Under **Option**, in the **Setting** list, select **On**.
6. Click **OK**.
7. Click **OK** to close the **Settings** dialog box.

You can also use the Assignment Editor to turn on the **Safe State Machine** option for specific state machines.

You can set the `syn_encoding safe` attribute on a state machine in HDL as shown in Example 8–25 through Example 8–27.

Example 8–25. Verilog HDL Example of a Safe State Machine Attribute

```
reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */;
```

Example 8–26. Verilog-2001 Example of a Safe State Machine Attribute

```
(* syn_encoding = "safe" *) reg [2:0] my_fsm;
```

Example 8–27. VHDL Example of a Safe State Machine Attribute

```
ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";
```

Safe state machine implementation can result in a noticeable area increase for the design. Therefore, Altera recommends that you set this option only on the critical state machines in the design where the safe mode is required, such as a state machine that uses inputs from asynchronous clock domains.

Note that if the `safe` state machine assignment is made on an instance that is not recognized as a state machine, or an entity that contains a state machine, the software takes no action. You must restructure the code so that the instance is recognized and properly inferred as a state machine.

For guidelines to ensure that your state machine is inferred correctly, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Level

This logic option causes a register (flip-flop) to power up with the specified logic level, either **High** (1) or **Low** (0). Registers in the device core hardware power up to 0 in all Altera devices. For the register to power up with a logic level **High** specified using this option, the Compiler performs an optimization referred to as NOT-gate push back on the register. NOT-gate push back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high and the device operates as expected. The register itself actually 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.

This 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

Quartus II integrated synthesis reads default values for registered signals defined in VHDL code and converts the default values into Power-Up Level settings. The software also synthesizes variables that are assigned values in Verilog HDL initial blocks into power-up conditions. Synthesis of these default and initial constructs enables the design's synthesized behavior to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

For more information about `NOT` gate push-back, the power-up states for Altera devices, and how power-up level is affected by set and reset control signals, refer to *Recommended HDL Coding Styles* 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 which do not have a defined power-up condition. This option is turned on by default.

For example, your design may have a register with its D input tied to V_{CC}, and with no clear signal or other secondary signals. If this option is enabled, the compiler can choose for the register to power up to V_{CC}. Therefore, the output of the register is always V_{CC}. The compiler can remove the register and connect its output to V_{CC}. If you turn this option off or if you set a Power-Up Level assignment of low for this register, the register transitions from GND to V_{CC} when the design starts up on the first clock signal. Thus, the register is not stuck at V_{CC} and cannot be

removed. Similarly, if the register has a clear signal it will not be removed, because after the clear is asserted, the register will again transition 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**.

Preserve Hierarchical Boundary

This logic option allows you to preserve the hierarchical boundaries between design entities. Beginning with the Quartus II software version 6.0, Altera recommends using design partitions incremental compilation instead of using the **Preserve Hierarchical Boundary** logic option.

The **Preserve Hierarchical Boundary** logic option may be removed in future versions of the Quartus II software.

Refer to "Partitions for Preserving Hierarchical Boundaries" on page 8–25 for details about using design partitions.

Remove Duplicate Registers

If you turn on this logic option, the Compiler removes registers that are identical to another register. If two registers generate the same logic, the Compiler removes the second one, and the first one fans out to the second one's destinations. Also, if the deleted register has different logic option assignments, the Compiler ignores them. This option is turned on by default.

Typically, you should use this option only if you want to prevent the Compiler from removing duplicate registers. That is, you should use this option only with the **Off** setting. You can apply this option to an individual register or a design entity that contains registers.

Remove Redundant Logic Cells

This logic option removes redundant `LCELL` primitives or WYSIWYG cells. The option is off by default to preserve logic cells that have been used intentionally. If you turn on this option, the Compiler optimizes a circuit for area and speed. You can set this option globally or apply it to individual nodes and entities. If you turn on the option at the global level, you can use the `keep` attribute or **Implement as Output of Logic Cell logic** option to preserve specific wire signals or nodes (refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 8–50).

Preserve Registers

This attribute and logic option direct 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. By default, the software may remove one of the two duplicate registers in this case; the `preserve` attribute can be added to both registers to prevent this.

This option cannot preserve registers that have no fan-out. To prevent the removal of registers with no fanout, refer to "Noprune Synthesis Attribute/Preserve Fanout Free Node" on page 8–49.

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 8–28, 8–29, and 8–30. In the 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 8–28. Verilog HDL Example of a syn_preserve Attribute

```
reg my_reg /* synthesis syn_preserve = 1 */;
```

Example 8–29. Verilog-2001 Example of a syn_preserve Attribute

```
(* syn_preserve = 1 *) reg my_reg;
```

The `" = 1"` after the `"preserve"` in Example 8–28 and Example 8–29 is optional, because the assignment uses a default value of `1` when it is specified.

Example 8–30. VHDL Example of a preserve Attribute

```
signal my_reg : stdlogic;
attribute preserve : boolean;
attribute preserve of my_reg : signal is true;
```

Noprune Synthesis Attribute/Preserve Fanout Free Node

This synthesis attribute and corresponding logic option direct the Compiler to preserve a fanout 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 registers' fanout logic is not specified. This option is supported for inferred registers in the Stratix and Cyclone series of devices, and MAX II devices.

You can set the **Preserve Fanout Free Node logic** option in the Quartus II GUI, or you can set the `noprune` attribute in your HDL code as shown in Example 8–31, Example 8–32, and Example 8–33. 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 fanout in its module or entity. If you do not use the synthesis attribute, registers with no fanout are removed (or "pruned") during analysis and elaboration before the logic synthesis stage applies any logic options. If the register has no fanout in the full design, but has fanout within its module or entity, then 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 8–31. Verilog HDL Example of a syn_noprune Attribute

```
reg my_reg /* synthesis syn_noprune = 1 */;
```

Example 8–32. Verilog-2001 Example of a noprune Attribute

```
(* noprune = 1 *) reg my_reg;
```

Example 8–33. VHDL Example of a noprune Attribute

```
signal my_reg : stdlogic;
attribute noprune: boolean;
attribute noprune of my_reg : signal is true;
```

Keep Combinational Node/Implement as Output of Logic Cell

This synthesis attribute and corresponding logic option direct the Compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. A wire that has a `keep` attribute or a node that has the **Implement as Output of Logic Cell** logic option applied becomes the output of a logic cell in the final synthesis netlist, and the name of the logic cell will be the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II 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*>`~reg0`).

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 8–34, Example 8–35, and Example 8–36. 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 8–34. Verilog HDL Example of a keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Example 8–35. Verilog-2001 Example of a keep Attribute

```
(* keep = 1 *) wire my_wire;
```

Example 8–36. VHDL Example of a syn_keep Attribute

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is 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 that contains these elements. You can use this option to reduce the load of critical signals, which can improve performance. You can use the option to instruct the Compiler to duplicate (or replicate) a register that feeds nodes in different locations on the target device. Duplicating the register may allow the Fitter to place these new registers closer to their destination logic, minimizing routing delay.

This option is available for all devices supported in the Quartus II software except MAX 3000, MAX 7000, FLEX 10K®, ACEX® 1K, and Mercury™ devices. To turn off the option for a given node if the option is set at a higher level of the design hierarchy, in the **Netlist Optimizations logic** option, select **Never Allow**. If not disabled by the **Netlist Optimizations** option, the maximum fan-out constraint is honored as long as the following conditions are met:

- The node is not part of a cascade, carry, or register cascade chain
- The node does not feed itself
- The node feeds other logic cells, DSP blocks, RAM blocks and/or pins through data, address, clock enable, etc, 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, in the third condition above where asynchronous control signals are involved, to avoid the possible situation that small differences in timing could produce functional differences in the implementation. If the constraint cannot be applied because one of these conditions is not met, the Quartus II software issues a message indicating that it ignored maximum fan-out assignment. To instruct the software not to check the node's destinations for possible problems like the third condition, you can set the **Netlist Optimizations logic** option to **Always Allow** for a given node.

If you have enabled any of the Quartus II netlist optimizations that affect registers, add the `preserve` attribute to any registers to which you have set a `maxfan` attribute. The `preserve` attribute ensures that the registers are not affected by any of the netlist optimization algorithms such as register retiming.

For details about netlist optimizations, refer to the *Netlist Optimization & 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, and this option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code as shown in Example 8–37, 8–38, and 8–39. In these examples, the Compiler duplicates the `clk_gen` register, so its fan-out is not greater than 50.

In addition to `maxfan`, the Quartus II software supports the `syn_maxfan` attribute name for compatibility with other synthesis tools.

Example 8–37. Verilog HDL Example of a syn_maxfan Attribute

```
reg clk_gen /* synthesis syn_maxfan = 50 */;
```

Example 8–38. Verilog-2001 Example of a maxfan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

Example 8–39. VHDL Example of a 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 & syn_direct_enable

The **Auto Clock Enable Replacement logic** option allows the software to find logic that feeds a register and move the logic to the register's clock enable input port. The option is on by default. You can set this option to **Off** for individual registers or design entities to solve fitting or performance issues with designs that have many clock enables. Turning the option off prevents the software from using the register's clock enable port, and 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 `syn_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 8–40 through Example 8–42 show how to set this attribute to ensure that the signal is preserved and used directly as a clock enable.

Example 8–40. Verilog HDL Example of a direct_enable attribute

```
wire my_enable /* synthesis syn_direct_enable = 1 */ ;
```

Example 8–41. Verilog-2001 Example of a direct_enable attribute

```
(* syn_direct_enable *) wire my_enable;
```

Example 8–42. VHDL Example of a direct_enable attribute

```
attribute syn_direct_enable: boolean;
attribute syn_direct_enable of my_enable: signal is true;
```

Megafunction Inference Control

The Quartus II Compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction The software uses the Altera megafunction code when compiling your design even when you do not specifically instantiate the megafunction. The software infers megafunctions to take advantage of logic that is optimized for Altera devices. The area and performance of such logic may 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, digital signal processing (DSP) blocks, and shift registers, that generally provide improved performance compared with basic logic elements.

For details on 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 sub-sections.

Multiply-Accumulators & Multiply-Adders

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the **Assignment Editor**.

Any registers that the software maps to the `altmult_accum` and `altmult_add` megafunctions and places in DSP blocks are not available in the Simulator because their node names do not exist after synthesis.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or for a specific block with the **Assignment Editor**. The software may not infer small shift registers because small shift registers typically do not benefit from implementation in dedicated memory. However, you can use the **Allow Any Shift Register Size for Recognition** logic option to instruct synthesis to infer a shift register even when its size is considered too small.

The registers that the software maps to the `altshift_taps` megafunction and places in RAM are not available in the Simulator because their node names do not exist after synthesis.

The **Auto Shift Register Replacement** logic option is turned off automatically when a formal verification tool is selected in the EDA Tool Settings. 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 black-box the shift register in a separate entity/module.

RAM & ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. These options are turned on by default. To disable inference, turn off the appropriate option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the **Assignment Editor**.

The software may not infer very small RAM or ROM blocks because very small memory blocks can typically be implemented more efficiently by using the registers in the logic. However, you can use the **Allow Any RAM Size for Recognition** and **Allow Any ROM Size for Recognition** logic options to instruct synthesis to infer a memory block even when its size is considered too small.

The **Auto ROM Replacement** logic option is automatically turned off when a formal verification tool is selected in the **EDA Tool Settings** page. A warning is issued and a report panel lists ROMs that would have been inferred if no formal verification tool was selected. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a ROM explicitly using the MegaWizard Plug-in Manager or create a black box the ROM in a separate entity/module.

Although formal verification tools do not support inferred RAM blocks, because of the importance of inferring RAM in many designs, the **Auto RAM Replacement** logic option remains on when a formal verification tool is selected in the **EDA Tool Settings** page. The Quartus II software automatically black boxes any module or entity that contains a RAM block that is inferred. The software issues a warning and lists the black box that is created in the compilation report. This block box allows formal verification tools to proceed; however, the entire module or entity containing the RAM cannot be verified in the tool. Altera recommends that you explicitly instantiate RAM blocks in separate modules or entities so that as much logic as possible can be verified by the formal verification tool.

RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** option allows the Quartus II integrated synthesis to convert RAM blocks that are small in size to logic cells if the logic cell implementation is deemed to give better quality of results. Only single-port or simple-dual port RAMs with no initialization files can be converted to logic cells. This option is off by default. You can set this option globally or apply it to individual RAM nodes.

For FLEX 10K, APEX, and the Stratix series of devices, the software use 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. Otherwise, use a RAM block if the total number of bits is greater than or equal to 32.

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 & ROM Style—for Inferred Memory

These attributes specify the implementation for an inferred RAM or ROM block. You can specify the type of TriMatrix™ embedded memory block to be used, or specify the use of standard logic cells (LEs or ALMs). The attributes are supported only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The values `"M512"`, `"M4K"`, `"M-RAM"`, `"MLAB"`, `"M9K"`, and `"M144K"` (as applicable for the target device family) indicates the type of memory block to use for the inferred RAM or ROM. The value `logic` indicates that the RAM or ROM should be implement 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 8–43, 8–44, and 8–45 specify that all memory in the module or entity `my_memory_blocks` should be implemented using a specific type of block.

Example 8–43. Verilog-1995 Example of Applying a romstyle Attribute to a Module Declaration

```
module my_memory_blocks (...) /* synthesis romstyle = "M4K" */
```

Example 8–44. Verilog-2001 Example of Applying a ramstyle Attribute to a Module Declaration

```
 (* ramstyle = "M512" *) module my_memory_blocks (...);
```

Example 8–45. VHDL Example of 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 8–46, 8–47, and 8–48 specify that the inferred memory `my_ram` or `my_rom` should be implemented using regular logic instead of a TriMatrix memory block.

Example 8–46. Verilog-1995 Example of Applying a syn_ramstyle Attribute to a Variable Declaration

```
reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */;
```

Example 8–47. Verilog-2001 Example of Applying a romstyle Attribute to a Variable Declaration

```
(* romstyle = "logic" *) reg [0:7] my_rom[0:63];
```

Example 8–48. VHDL Example of Applying a ramstyle Attribute to a Signal Declaration

```
type memory_t is array (0 to 63) of std_logic_vector (0 to 7);
signal my_ram : memory_t;
attribute ramstyle : string;
attribute ramstyle of my_ram : signal is "logic";
```

Turning off Add Pass-Through Logic to Inferred RAMs/ no_rw_check

The `no_rw_check` value for the ramstyle attribute, and the corresponding global logic option **Add Pass-Through Logic to Inferred RAMs** indicate that your design does not depend on the behavior of the inferred RAM when there are simultaneous reads and writes to the same address. If you specify either the attribute or the logic option, the Quartus II software can choose a read-during-write behavior instead of using the read-during-write behavior of your HDL source code.

In some cases, an inferred RAM must be mapped into regular logic cells because it has a read-during-write behavior that is not supported by the TriMatrix memory blocks in your target device. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source, increasing the area of your design and potentially reducing its performance. In these cases, you can use the attribute to specify that the software can implement the RAM directly in a TriMatrix memory block without using logic. You can also use the attribute to prevent a warning message for dual-clock RAMs in the case that the inferred behavior in the device does not exactly match the read-during-write conditions described in the HDL code.

For more information about recommended styles for inferring RAM and some of the issues involved in read-during-write conditions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To set the **Add Pass-Through Logic to Inferred RAMs** logic option through the Quartus II GUI, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. RAM Example 8–49 and 8–50 use two addresses and normally require extra logic after the RAM in order to ensure that the read-during-write conditions in the device match the HDL code. If you don't require a defined read-during-write condition in your design, this extra logic is not required. With the `no_rw_check` attribute, Quartus II integrated synthesis won't generate the extra logic.

Example 8–49. 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 8–50. 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;
```

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory in the form of a Memory Initialization File (**.mif**). The attribute takes a string value containing the name of the RAM initialization file.

Example 8–51. Verilog-1995 Example of Applying a ram_init_file Attribute

```
reg [7:0] mem[0:255] /* synthesis ram_init_file
= " my_init_file.mif" */;
```

Example 8–52. Verilog-2001 Example of Applying a ram_init_file Attribute

```
(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];
```

Example 8–53. VHDL Example of 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.

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, among other things, on the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

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.

In addition to `multstyle`, the Quartus II software supports the `syn_multstyle` attribute name for compatibility with other synthesis tools.

When applied to a Verilog HDL module declaration, the attribute specifies the default implementation style for all instances of the `*` operator in the module. For example, in the following code examples, the `multstyle` attribute directs the Quartus II software to implement all multiplications inside module `my_module` in dedicated multiplication hardware.

Example 8–54. Verilog-1995 Example of Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

Example 8–55. Verilog-2001 Example of Applying a multstyle Attribute to a Module Declaration

```
(* multstyle = "dsp" *) module my_module(...);
```

When applied to a Verilog HDL variable declaration, the attribute specifies the implementation style to be used for a multiplication operator whose result is directly assigned to the variable. It overrides the `multstyle` attribute associated with the enclosing module, if present. For example, in Example 8–56 and 8–57, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 8–56. Verilog-2001 Example of 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 8–57. Verilog-1995 Example of 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. For example, in Example 8–58, the `multstyle` attribute indicates that `a * b` should be implemented in dedicated hardware.

Example 8–58. Verilog-2001 Example of Applying a multstyle Attribute to a Binary Expression

```
wire [8:0] a, b;
wire [17:0] result;
assign result = a * (* multstyle = "dsp" *) b;
```

You can not 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. For example, in Example 8–59, 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 8–59. VHDL Example of Applying a multstyle Attribute to an Architecture

```
architecture rtl of my_entity is
       attribute multstyle : string;
       attribute multstyle of rtl : architecture is "dsp";
begin
```

When applied to a VHDL signal or variable, the attribute specifies the implementation style to be used for all instances of the * operator whose result is directly assigned to the signal or variable. It overrides the multstyle attribute associated with the enclosing entity or architecture, if present. For example, in Example 8–60, the multstyle attribute associated with signal result directs the Quartus II software to implement a * b in general logic rather than dedicated hardware.

Example 8–60. VHDL Example of Applying a multstyle Attribute to a Signal or Variable

```
signal a, b : unsigned(8 downto 0);
signal result : unsigned(17 downto 0);

attribute multstyle : string;
attribute multstyle of result : signal is "logic";
result <= a * b;
```

Full Case

A Verilog HDL case statement is considered full when its case items cover all possible binary values of the case expression or when a default case statement is present. A full_case attribute attached to a case statement header that is not full forces the unspecified states to be treated as a "don't care" value. VHDL case statements must be full, so the attribute does not apply to VHDL.

Using this attribute on a case statement that is not full avoids the latch inference problems discussed in the *Design Recommendations for Altera Devices* 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, e.g. 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 8–32).

When you are using the `full_case` attribute, there is a potential cause for a simulation mismatch between Verilog HDL functional and post-Quartus II simulation because unknown case statement cases may still function like latches during functional simulation. For example, a simulation mismatch may occur with the code in the following example when `sel` is `2'b11` because a functional HDL simulation output behaves like a latch while the Quartus II simulation output behaves like "don't care."

Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

The case statement in Example 8–61 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 8–61. Verilog HDL Example of 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 8–62 in the `case` header instead of the comment form shown in Example 8–61.

Example 8–62. 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 may overlap. To resolve multiple matching case items, the Verilog HDL language defines a priority relationship among case items in which the case statement always executes the first case item that matches the case expression value. By default, the Quartus II software implements the extra logic required to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement's header allows the Quartus II software to consider its case items as inherently parallel, that is, at most one case item matches the case expression value. Parallel case items reduce the complexity of the generated logic.

In VHDL, the individual choices in a case statement may not overlap, so they are always parallel and this attribute does not apply.

Use this attribute only when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic will not match the functional simulation behavior of the Verilog HDL.

Altera recommends that you avoid using the `parallel_case` attribute, due to the possibility of introducing mismatches between Verilog HDL functional and post-Quartus II simulation.

If you specify the supported Verilog HDL version as SystemVerilog-2005 for your design, you can use the SystemVerilog keyword `unique` to achieve the same result as the `parallel_case` directive without causing simulation mismatches.

The following example 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 may simulate differently because the `parallel_case` attribute eliminates this priority order. If more than one bit of `sel` is high, then more than one output (`a`, `b`, or `c`) is high as well, a situation that cannot occur in functional HDL simulation.

Example 8–63. Verilog HDL Example of 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 as shown in Example 8–64 in the `case` (or `casez`) header instead of the comment form as shown in Example 8–63.

Example 8–64. Verilog-2001 Syntax

```
(* parallel_case *) casez (sel)
```

Translate Off & On / Synthesis Off & 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 8–65 and Example 8–66 show these directives.

Example 8–65. Verilog HDL Example of Translate Off & On

```
// synthesis translate_off
parameter tpd = 2;    // Delay for simulation
#tpd;
// synthesis translate_on
```

Example 8–66. VHDL Example of Translate Off & On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

If you wish to ignore a portion of code in Quartus II integrated synthesis only, you can use the Altera-specific attribute keyword `altera`. For example, use the `// altera translate_off` and `// altera translate_on` directives to direct Quartus II integrated synthesis to ignore a portion of code that is intended only for other synthesis tools.

Ignore translate_off and synthesis_off Directives

The **Ignore translate_off and synthesis_off directives** logic option directs Quartus II integrated synthesis to ignore the `translate_off` and `synthesis_off` directives described in the previous section. This allows you to compile code that was previously intended to be ignored by third-party synthesis tools, for example megafunction declarations that were treated as black boxes in other tools but can be compiled in the Quartus II software. To set the **Ignore translate_off and synthesis_off directives** logic option, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

Read Comments as HDL

The `read_comments_as_HDL` synthesis directive indicates that the Quartus II software should compile a portion of HDL code that is commented out. This directive allows you to comment out portions of HDL source code that are not relevant for simulation, while instructing the Quartus II software to read and synthesize that same source code. Setting the `read_comments_as_HDL` directive to `on` marks the beginning of commented code that the synthesis tool should read; setting the `read_comments_as_HDL` directive to `off` indicates the end of the code.

You can use this directive with `translate_off` and `translate_on` to create one HDL source file that includes both a megafunction instantiation for synthesis and a behavioral description for simulation.

Because formal verification tools do not recognize the `read_comments_as_HDL` directive, it is not supported when you are using formal verification.

In Example 8–67 and 8–68, the commented code enclosed by `read_comments_as_HDL` is visible to the Quartus II Compiler and is synthesized.

Because synthesis directives are case-sensitive in Verilog HDL, you must match the case of the directive, as shown below.

Example 8–67. 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 8–68. 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
```

Use I/O Flip-Flops

This attribute directs the Quartus II software to implement input, output, and output enable flip-flops (or registers) in I/O cells that have fast, direct connections to an I/O pin, when possible. Applying the `useioff` synthesis attribute can improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times. This synthesis attribute is supported using the **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options that can also be set in the **Assignment Editor**.

For more information about which device families support fast input, output, and output enable registers, refer to the device family data sheet, device handbook, or the Quartus II Help.

The `useioff` synthesis attribute takes a Boolean value and can only be applied to the port declarations of a top-level Verilog HDL module or VHDL entity (it is ignored if applied elsewhere). Setting the value to `1` (Verilog HDL) or `TRUE` (VHDL) instructs the Quartus II software to pack registers into I/O cells. Setting the value to 0 (Verilog HDL) or `FALSE` (VHDL) prevents register packing into I/O cells.

In Example 8–69 and 8–70, 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 8–69. Verilog HDL Example of a 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 8–70 and 8–71 instead of the comment form shown in Example 8–69.

Example 8–70. Verilog-2001 Syntax for a useioff Attribute

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Example 8–71. VHDL with a useioff Attribute

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity top_level 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 top_level;
architecture rtl of top_level is
   signal o_reg, a_reg, b_reg : unsigned(1 downto 0);
begin
   process(clk)
   begin
      a_reg <= a;
      b_reg <= b;
      o_reg <= a_reg + b_reg;
   end process;
   o <= o_reg;
end rtl;
```

Specifying Pin Locations with chip_pin

This attribute enables you to assign pins to the ports of an entity or module in your HDL source. You may only assign pins to single-bit or one-dimensional bus ports in your design.

For single-bit ports, the value of the `chip_pin` attribute is the name of the pin on the target device, as specified by the device's pin table.

In addition to `chip_pin`, the Quartus II software supports the `altera_chip_pin_lc` attribute name for compatibility with other synthesis tools. When using this attribute in other synthesis tools, some older device families require an "@" symbol in front of each pin assignment. In the Quartus II software, the "@" is optional.

Example 8–72, 8–73, and 8–74 show different ways of assigning input pin `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

Example 8–72. Verilog-1995 Examples of Applying Chip Pin to a Single Pin

```
input my_pin1 /* synthesis chip_pin = "C1" */;
input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
```

Example 8–73. Verilog-2001 Example of Applying Chip Pin to a Single Pin

```
(* chip_pin = "C1" *) input my_pin1;
(* altera_chip_pin_lc = "@4" *) input my_pin2;
```

Example 8–74. VHDL Example of Applying Chip Pin to a Single Pin

```
entity my_entity is
     port(my_pin1: in std_logic; my_pin2: in std_logic;…);
end my_entity;
attribute chip_pin : string;
attribute altera_chip_pin_lc : string;
attribute chip_pin of my_pin1 : signal is "C1";
attribute altera_chip_pin_lc of my_pin2 : signal is "@4"
```

For bus I/O ports, the value of the chip pin attribute is a comma-delimited list of pin assignments. The order in which you declare the port's range determines the mapping of assignments to individual bits in the port. To leave a particular bit unassigned, simply leave its corresponding pin assignment blank.

Example 8–75 assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

Example 8–75. Verilog-1995 Example of Applying Chip Pin to a Bus of Pins

```
input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

Example 8–76 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 8–76. Verilog-1995 Example of Applying Chip Pin to Part of a Bus

```
input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;
```

Example 8–77 assigns `my_pin[2]` to Pin `4` and `my_pin[0]` to Pin `6`, but leaves `my_pin[1]` unassigned.

Example 8–77. VHDL Example of Applying Chip Pin to Part of a Bus of Pins

```
entity my_entity is
     port(my_pin: in std_logic_vector(2 downto 0);…);
end my_entity;

attribute chip_pin of my_pin: signal is "4, , 6";
```

Using Altera Attribute to Set Quartus II Logic Options

This attribute enables you to apply Quartus II options and assignments to an object (entity, instance, or net) in your HDL source code. 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 user interface, the Quartus II Settings File (**.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 8–32.

The attribute value is a single string containing a list of Quartus II Settings File variable assignments separated by semicolons, as shown in the following example:

```
-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 following syntax for each Quartus II Settings File variable assignment.

```
-from <source> -to <target> -section_id <section>
-name <variable> <value>
```

The syntax for the full attribute value, including the optional target, source, and section tags for two different Quartus II Settings File assignments is shown in the following example:

"[-from <*source_1*>] [-to <*target_1*>] [-section_id <*section_1*>] -name <*variable_1*> <*value_1*>; [-from <*source_2*>] [-to <*target_2*>] [-section_id <*section_2*>] -name <*variable_2*> <*value_2*>"

If a variable's assigned value is a string of text, you must use escaped quotes around the value, as in the following examples (using non-existent variable and value terms):

Verilog HDL

```
"VARIABLE_NAME \"STRING_VALUE\""
```

VHDL

```
"VARIABLE_NAME ""STRING_VALUE"""
```

To find the Quartus II Settings File variable name or value corresponding to a specific Quartus II option or assignment, you can make the option setting or assignment in the Quartus II user interface 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 8–78, 8–79, and 8–80 use `altera_attribute` to set the power-up level of an inferred register. Note that for inferred instances, you cannot apply the attribute to the instance directly so you should apply the attribute to one of the instance's output nets. The Quartus II software moves the attribute to the inferred instance automatically.

Example 8–78. Verilog-1995 Example of Applying Altera Attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH" */;
```

Example 8–79. Verilog-2001 Example of Applying Altera Attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;
```

Example 8–80. VHDL Example of 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 8–81, 8–82, and 8–83 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 8–81. Verilog-1995 Example of Applying Altera Attribute to an Entity

```
module my_entity(…) /* synthesis altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Example 8–82. Verilog-2001 Example of Applying Altera Attribute to an Entity

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *)
module my_entity(…) ;
```

Example 8–83. VHDL Example of 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 8–84, 8–85, and 8–86, the `altera_attribute` is used to cut all timing paths from `reg1` to `reg2`, equivalent to this Tcl or QSF command:

```
set_instance_assignment -name CUT ON -from reg1 -to reg2
```

Example 8–84. Verilog-1995 Example of Applying Altera Attribute with -to

```
reg reg2;
reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;
```

Example 8–85. Verilog-2001 Example of Applying Altera Attribute with -to

```
reg reg2;
(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;
```

Example 8–86. VHDL Example of Applying Altera Attribute with -to

```
signal reg1, reg2 : std_logic;
attribute altera_attribute: string;
attribute altera_attribute of reg1 : signal is "-name CUT ON -to reg2";
```

You may 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 may also specify wildcards for either option. For example, if you specify "`*`" for the `-to` option instead of `reg2` in these examples, the Quartus II software cuts all timing paths from `reg1` to every other register in this design entity.

The `altera_attribute` can be used only for entity-level settings, and the assignments (including wildcards) apply only to the current entity.

Analyzing Synthesis Results

After you have performed synthesis, you can check your synthesis results in the following locations:

- Messages
- Analysis & Synthesis Section of Compilation Report
- Project Navigator

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. You should always check the messages to analyze critical warnings and warnings, because these messages may relate to important design problems. It is also useful to read the Information messages to get more information about how the software processes your design.

Analysis & Synthesis Section of Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation, or you can choose it from the Processing menu. After Analysis & Synthesis, before the Fitter begins, the Summary information provides a summary of utilization based on

synthesis data, before fitter optimizations have occurred. Synthesis-specific information is listed in the **Analysis & Synthesis** section.

There are various report sections under Analysis & Synthesis, including a list of the source files read for the project, the resource utilization by entity after synthesis, and information about state machines, latches, optimization results, and parameter settings.

For more information about each report section, refer to the Quartus II Help.

Project Navigator

The Hierarchy tab of the Project Navigator provides a summary of resource information about the entities in the project. After Analysis & 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 that shows parameter information for each instance.

VHDL & 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 help you identify potential problems early in the design process.

HDL Message Types

HDL messages fall into three categories: Info, Warning, and Error.

- **Info message**—Lists a property of your design.
- **Warning message**—Indicates a potential problem in your design. Potential problems come from a variety of sources, including typos, inappropriate design practices, or the functional limitations of your target device. Though HDL warning messages do not always identify actual problems, you should always investigate code that generates an HDL warning. Otherwise, the synthesized behavior of your design might not match your original intent or its simulated behavior.
- **Error message**—Indicates an actual problem with your design. Your HDL code may be invalid due to a syntax or semantic error, or it may 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 8–87, the sensitivity list contains multiple copies of the variable `i`. While the Verilog HDL language does not prohibit duplicate entries in a sensitivity list, it is clear that this design has a typo: Variable `j` should be listed on the sensitivity list to avoid a possible simulation/synthesis mismatch.

Example 8–87. Generating an HDL Warning Message

```
//dup.v
module dup(input i, input j, output reg o);
always @ (i or i)
    o = i & j;
endmodule
```

When processing this HDL code, the Quartus II software generates the following warning message:

```
Warning: (10276) Verilog HDL sensitivity list warning
at dup.v(2): sensitivity list contains multiple
entries for "i".
```

In Verilog HDL, variable names are case-sensitive, so the variables my_reg and MY_REG in Example 8–88 are two different variables. However, declaring variables whose names only differ in case may confuse some users, especially those users who use VHDL, where variables are not case-sensitive.

Example 8–88. Generating HDL Info Messages

```
// namecase.v
module namecase (input i, output o);
    reg my_reg;
    reg MY_REG;
    assign o = i;
endmodule
```

When processing this HDL code, the Quartus II software generates the following informational message:

```
Info: (10281) Verilog HDL information at
namecase.v(3): variable name "MY_REG" and variable
name "my_reg" should not differ only in case.
```

In addition, the Quartus II software generates additional HDL info messages to inform you that neither my_reg or MY_REG are used in this small design:

```
Info: (10035) Verilog HDL or VHDL information at
namecase.v(3): object "my_reg" declared but not used
Info: (10035) Verilog HDL or VHDL information at
namecase.v(4): object "MY_REG" declared but not used
```

Controlling the Display of HDL Messages

The Quartus II software allows you to control how many HDL messages you see during the analysis and elaboration of your design files. You can set the HDL Message Level to enable or disable groups of HDL messages, or you can enable or disable specific messages.

For more information about synthesis directives and their syntax, refer to "Synthesis Directives" on page 8–35.

These options are in addition to the general message suppression options in the Quartus II software. For more information about suppressing other messages, refer to the *Quartus II Project Management* chapter in volume 2 of the *Quartus II Handbook*.

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 8–6 details the information about the HDL message levels.

Table 8–6. HDL Info Message Level

Level	Purpose	Description
Level1	Displays high-severity messages only	If you want to see only those HDL messages that identify likely problems with your design, select **Level1**. When **Level1** is selected, the Quartus II software issues a message only if there is a high probability that it points an actual problem with your design.
Level2	Displays high-severity and medium-severity messages	If you want to see additional HDL messages that identify possible problems with your design, select **Level2**. This is the default setting.
Level3	Displays all messages, including low-severity message	If you want to see all HDL info and warning messages, select **Level3**. This level includes extra "LINT" messages that suggest changes to improve the style of your HDL code or make it easier to understand.

You should address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the user interface, on the Assignments menu, click **Settings**; under Category, click **Analysis & Synthesis Settings**. Set the HDL Message Level.

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 8–89 and 8–90.

Example 8–89. Verilog HDL Examples of message_level Directive

```
// altera message_level level1
```

or

```
/* altera message_level level3 */
```

Example 8–90. VHDL Example of 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

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.

To disable specific HDL messages in the GUI, from the Settings menu, select **Analysis & Synthesis Settings** and click the **Advanced** button next to the HDL Message Level setting. In the Advanced Message Settings dialog box, add the Message IDs you wish to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. Both directives take a space-separated list of Message IDs. You can enable or disable messages with these synthesis directives immediately before Verilog HDL modules, VHDL entities, or VHDL architectures. You cannot enable or disable a message in the middle of an HDL construct.

A message enabled or disabled via a `message_on` or `message_off` synthesis directive overrides its HDL Message Level or any `message_level` synthesis directive. The message will remain disabled until the end of the source file or until its status is changed by another `message_on` or `message_off` directive.

Example 8–91. Verilog HDL message_off Directive for Message with ID 10000

```
// altera message_off 10000
```

or

```
/* altera message_off 10000 */
```

Example 8–92. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

Being able to find the logic node names after synthesis can be useful during verification or while debugging a design. This section provides an overview of the conventions used by the Quartus II software when it names the nodes created from your HDL design. The section focuses on the conventions for Verilog HDL and VHDL code, but AHDL and BDFs are discussed when appropriate.

Whenever possible, as described in this section, Quartus II integrated synthesis uses wire or signal names from your source code to name nodes such as LEs or ALMs. Some nodes, such as registers, have predictable names that typically do not change when a design is resynthesized. The names of other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.

Synthesis netlist optimizations also change node names because nodes are changed, combined, and duplicated to optimize the design.

For more information about the type of optimizations performed by synthesis netlist optimizations, refer to *Netlist Optimizations & Physical Synthesis* in volume 2 of the *Quartus II handbook*.

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.

Hierarchical Node-Naming Conventions

To make each name in the design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator is used to indicate a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, using the ":" separator between each entity name and its instance name. For example, if a design instantiates entity `A` with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. The full name of any node is obtained by starting with the hierarchical instance path; followed by a "|", and ending with the node name inside that entity, using the following convention:

<entity 0>`:`*<instance_name 0>*`|`*<entity 1>*:
<instance_name 1>`|...|`*<instance_name n>*

For example, if entity A contains a register (DFF atom) called `my_dff`, its full hierarchy name would be `A:my_A_inst|my_dff`.

You can turn off the **Display Entity Name for Node Name** option on the **Compilation Process Settings** page of the **Settings** dialog box to instruct the Compiler to generate node names that do not contain the name for each level of the hierarchy. With this option on, the node names use the following convention:

<instance_name 0>|*<instance_name 1>*|...|*<instance_name n>*

Node-Naming Conventions for Registers (DFF or D Flip-Flop Atoms)

In Verilog HDL and VHDL, inferred registers are named after the `reg` or `signal` connected to the output.

For example, the following is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 8–93. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

always @ (posedge clk)
  my_dff_out <= dff_in;
```

Similarly, Example 8–94 is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 8–94. 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 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 above examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (e.g., cannot use

my_dff_out) because the software requires that all logic and I/O cells have unique names. In this case, the Quartus II integrated synthesis appends ~reg0 to the register name.

For example, the Verilog HDL code in Example 8–95 produces a register called q~reg0:

Example 8–95. 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, then 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 may not be able to find registers that you expect to see in the synthesis netlist. Registers may be removed by logic optimization, or their names may 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 device features that affect register names:

- State machines
- Inferred adder-subtractors, shift registers, memory, and DSP functions
- Input and output registers of RAM and DSP blocks

State Machines

If a state machine is inferred from your HDL code, then the registers that represent the states will be mapped into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form where each state is represented by one register. In this case for Verilog HDL or VHDL designs, the registers are named according to the name of the state register and the states, where possible.

For example, consider a Verilog HDL state machine where the states are `parameter state0 = 1, state1 = 2, state2 = 3,` and where the state machine register is declared as `reg [1:0] my_fsm`. In this example, the three one-hot state registers are named `my_fsm.state0`, `my_fsm.state1`, and `my_fsm.state2`.

In AHDL, state machines are explicitly specified with a machine name. State machine registers are given synthesized names based on the state machine name but not the state names. For example, if a state machine is called `my_fsm` and has four state bits, they may 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 & DSP Functions

The Quartus II software infers megafunctions from Verilog HDL and VHDL code for logic that forms adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that can be placed in DSP blocks.

For information about inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in the *Quartus II Handbook*.

Because adder-subtractors are part of a megafunction instead of generic logic, the combinational logic exists in the design with different names. For shift registers, memory, and DSP functions, the registers and logic are typically implemented inside the dedicated RAM or DSP blocks in the device. Thus, the registers are not visible as separate LEs or ALMs.

Input & Output Registers of RAM & DSP Blocks

Registers can be packed into the input registers and output registers of RAM and DSP blocks, so that they are not visible as separate registers in LEs or ALMs.

For information about packing registers into RAM and DSP megafunctions, refer to the *Recommended HDL Coding Styles* chapter in the *Quartus II Handbook*.

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 may change the node names due to synthesis optimizations.

For example, consider the Verilog HDL code in Example 8–96. Quartus II integrated synthesis uses the names `c`, `d`, `e`, and `f` for the combinational logic cells that are produced.

Example 8–96. 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 BDF, all elements are given a name when they are instantiated in the design and the software uses the user-defined name when possible.

If logic cells, such as those created in the above example, are packed with registers in device architectures such as the Stratix and Cyclone device families, those names may 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 above. In some cases, synthesized names will be used, which are the wire names with a tilde (~) and a number appended. For example, if a complex expression is assigned to a wire `w` and that expression generates several logic cells, those cells may have names such as `w`, `w~1`, `w~2`, and so on. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. It is a goal of Quartus II integrated synthesis to retain user names whenever possible. Any node name ending with ~<*number*> is a name created during synthesis, which may change if the design is

changed and re-synthesized. Knowing these naming conventions can help you understand your post-synthesis results and make it easier to debug your design or make assignments.

Scripting Support

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

```
quartus_sh --qhelp ↵
```

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

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

You can specify many of the options described in this section either on an instance, or global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF Variable Name> <Value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF Variable Name> <Value>\
-to <Instance Name>
```

Adding an HDL File to a Project & Setting the HDL Version

Use the following Tcl assignments to add an HDL or schematic entry design file to your project:

```
set_global_assignment -name VERILOG_FILE <file name>.<v|sv>
set_global_assignment -name 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 header files.

To specify the Verilog HDL or VHDL version, use the following option at the end of the VERILOG_FILE or VHDL_FILE command:

-HDL_VERSION <*language version*>

The variable <*language version*> takes one of the following values:

- VERILOG_1995
- VERILOG_2001
- SYSTEMVERILOG_2005
- VHDL87
- VHDL93

For example, to add a Verilog HDL file called **my_file** that is written in SystemVerilog-2005, use the following command:

```
set_global_assignment -name VERILOG_FILE my_file.v -HDL_VERSION SYSTEMVERILOG_2005
```

Quartus II Synthesis Options

Table 8–7 lists the Quartus II Settings File variable names and applicable values for the settings discussed in this chapter. The Quartus II Settings 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, or an Instance setting, or both.

Table 8–7. Quartus II Synthesis Options (Part 1 of 2)

Setting Name	Quartus II Settings File Variable	Values	Type
Allow Any RAM Size for Recognition	ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Allow Any ROM Size for Recognition	ALLOW_ANY_ROM_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Allow Any Shift Register Size for Recognition	ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Auto DSP Block Replacement	AUTO_DSP_RECOGNITION	ON, OFF	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
Fast Input Register	FAST_INPUT_REGISTER	ON, OFF	Instance

Table 8–7. Quartus II Synthesis Options (Part 2 of 2)

Setting Name	Quartus II Settings File Variable	Values	Type
Fast Output Enable Register	FAST_OUTPUT_ENABLE_REGISTER	ON, OFF	Instance
Fast Output Register	FAST_OUTPUT_REGISTER	ON, OFF	Instance
Implement as Output of Logic Cell	IMPLEMENT_AS_OUTPUT_OF_LOGIC_CELL	ON, OFF	Instance
Maximum Fan-Out	MAX_FANOUT	*<Maximum Fan-Out Value>*	Instance
Optimization Technique	*<device family>*_OPTIMIZATION_TECHNIQUE	Area, Speed, Balanced	Global, Instance
PowerPlay Power Optimization	OPTIMIZE_POWER_DURING_SYNTHESIS	"NORMAL COMPILATION", "EXTRA EFFORT", OFF	Global, Instance
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
Power-Up Level	POWER_UP_LEVEL	HIGH, LOW	Instance
Preserve Hierarchical Boundary	PRESERVE_HIERARCHICAL_BOUNDARY	Off, Relaxed, Firm	Instance
Preserve Registers	PRESERVE_REGISTER	ON, OFF	Instance
Remove Duplicate Logic	REMOVE_DUPLICATE_LOGIC	ON, OFF	Global, Instance
Remove Duplicate Registers	REMOVE_DUPLICATE_REGISTERS	ON, OFF	Global, Instance
Remove Redundant Logic Cells	REMOVE_REDUNDANT_LOGIC_CELLS	ON, OFF	Global
Restructure Multiplexers	MUX_RESTRUCTURE	On, Off, Auto	Global, Instance
Speed Optimization Technique for Clock Domains	SYNTH_CRITICAL_CLOCK	ON, OFF	Instance
State Machine Processing	STATE_MACHINE_PROCESSING	AUTO, "MINIMAL BITS", "ONE HOT", "USER-ENCODED"	Global, Instance

Assigning a Pin

Use the following Tcl command to assign a signal to a pin or device location.

```
set_location_assignment -to <signal name> <location>
```

For example,
```
set_location_assignment -to data_input Pin_A3
```

Valid locations are pin location names. Some device families also support edge and I/O bank locations. Edge locations are `EDGE_BOTTOM`, `EDGE_LEFT`, `EDGE_TOP`, and `EDGE_RIGHT`. I/O bank locations include `IOBANK_1` to `IOBANK_n`, where n is the number of I/O banks in a particular device.

Preparing a Design for Incremental Synthesis

To set up your design for incremental synthesis, identify design partitions and enable incremental synthesis.

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, 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 8–80.

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

The <*file name*> is the name used for internally generated 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, and you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

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

Enabling Incremental Synthesis

Turn on incremental synthesis using the following Tcl command:

```
set_global_assignment -name INCREMENTAL_COMPILATION \
INCREMENTAL_SYNTHESIS
```

Synthesizing a Design Using Incremental Synthesis

Once incremental synthesis is turned on in the Quartus II Settings File file, or through a Tcl command, incremental synthesis automatically occurs when you compile using the `execute_flow -compile` command for the **quartus_sh** compiler executable.

Synthesizing Using the Synthesis & Merge Commands
Use the separate synthesis and merge commands if you compile your design using the individual compiler executables (e.g., **quartus_map** and **quartus_fit**) instead of using the `execute_flow -compile` command for the **quartus_sh** compiler executable.

To enable incremental synthesis when using the **quartus_map** executable, perform the following two steps:

1. Type the following command at a command prompt:

```
quartus_map --incremental compilation=incremental_synthesis ↵
```

This command enables the flow in the project, so subsequent synthesis runs can be performed with the `quartus_map` command without the incremental compilation option.

2. Merge the synthesized partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting. Type the following command at a system command prompt:

```
quartus_cdb --merge ↵
```

Conclusion

The Quartus II software includes complete Verilog HDL and VHDL language support, as well as support for Altera-specific languages, making it an easy-to-use, standalone solution for Altera designs. You can use the synthesis options available in the software to help you improve your synthesis results, giving you more control over the way your design is synthesized.

Document Revision History

Table 8–8 shows the revision history for this document.

Table 8–8. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	• Added information on how to set the HDL version in "Verilog HDL Support" on page 8–5 and "VHDL Support" on page 8–10 • Updated the list of supported constructs in "SystemVerilog Support" on page 8–7 • Added "Initial Constructs & Memory System Tasks" on page 8–8 • Added "Design Libraries" on page 8–13 to include information on libraries and duplicate entity names in all languages • Added "Using Parameters/Generics" on page 8–18 • Reorganized the options in the Quartus II Synthesis Options section • Added information about reset status to "State Machine Processing" on page 8–40 • Added "Safe State Machines" on page 8–43 • Removed section on obsoleted logic option Remove Duplicate Logic • Added "Controlling Clock Enable Signals with Auto Clock Enable Replacement & syn_direct_enable" on page 8–52 • Added "RAM to Logic Cell Conversion" on page 8–55 • Added "Turning off Add Pass-Through Logic to Inferred RAMs/ no_rw_check" on page 8–57 • Added synthesis_off and on directives to "Translate Off & On / Synthesis Off & On" on page 8–65 and "Ignore translate_off and synthesis_off Directives" on page 8–66 • Updated options to include Stratix III in the Stratix series of devices as required	This chapter has been updated to include information about additional functionality and support for integrated synthesis. The updates made to this chapter describe new and/or enhanced features to language support, incremental synthesis, and many of the Quartus II synthesis options.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Added language support. • Added Quartus II Synthesis options. • Added information on setting other Quartus II options in HDL source code.	
December 2005 v5.1.1	Minor typographic update.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 7 was formerly Chapter 8 in version 5.0.	

May 2005 v5.0.0	• Chapter 8 was formerly Chapter 6 in version 4.2. • Updated information. • Updated figures. • Restructured information. • Renamed sections. • New functionality for Quartus II software 5.0.	
Dec. 2004 v3.0	• Chapter 7 was formerly Chapter 8 in version 4.1. • Added documentation of incremental synthesis feature • New functionality for Quartus II software version 4.2	
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software version 4.1.	
Feb. 2004 v1.0	Initial release.	

9. Synplicity Synplify & Synplify Pro Support

QII51009-6.1.0

Introduction

As programmable logic device (PLD) designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. This chapter documents support for the Synplicity Synplify and Synplify Pro 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 in a clear- or black-box flow 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 Synplify Pro software MultiPoint flow

The content in this chapter applies to both the Synplify and Synplify Pro software unless otherwise specified.

This chapter assumes that you have set up, licensed, and are familiar with the Synplify or Synplify Pro software.

For more information about obtaining, licensing, and using the Synplify software, refer to the Synplicity web site at **www.synplicity.com**.

Altera Device Family Support

The Synplify software maps synthesis results to Altera device families. The following list shows the Altera device families supported by the Synplify software version 8.8, that can be compiled in the Quartus II software version 6.1:

- Stratix® III
- Stratix II, Stratix II GX, Hardcopy® II
- Stratix, Stratix GX, HardCopy Stratix

- MAX® II
- Cyclone® II
- Cyclone
- MAX® 7000, MAX 3000
- APEX™ II
- APEX 20K, APEX 20KC, APEX 20KE
- FLEX® 10K, FLEX 6000
- ACEX® 1K

The Synplify software also supports the following legacy devices that are supported in the Quartus II software only with a specific license requested at www.altera.com/mysupport:

- Excalibur™ -ARM®
- Mercury™

In addition, the Synplify software supports the following legacy devices that are supported only in the Altera MAX+PLUS II software:

- FLEX 8000
- MAX 9000

To learn about new device support for a specific Synplify version, refer to the release notes on Synplicity's web site at **www.synplicity.com.**

Design Flow

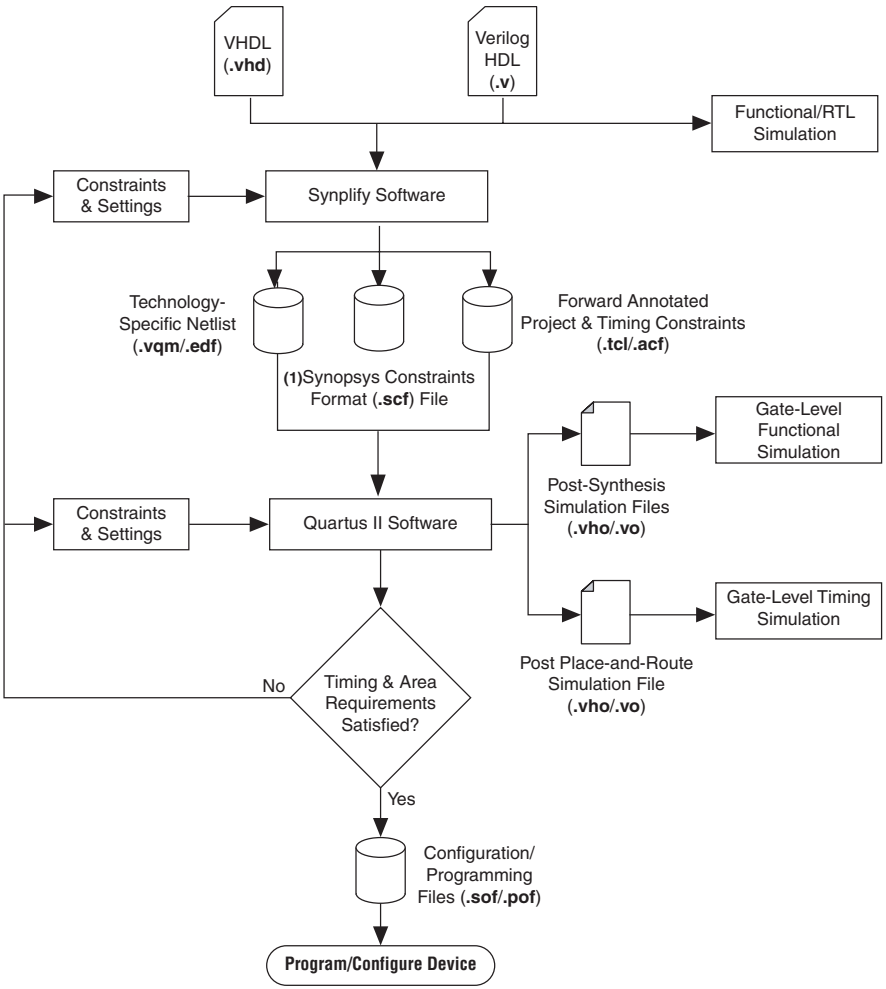
A Quartus II software design flow using the Synplify software consists of the following steps:

1. Create Verilog HDL or VHDL design files in the Quartus II software, Synplify software, or a text editor.
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 to optimize the design during synthesis.
4. Create a Quartus II project and import the technology-specific EDIF (**.edf**) or VOM (**.vqm**) netlist, the Synopsys Constraints Format (**.scf**) file (for TimeQuest constraints if Stratix III is selected), and the tool command language (Tcl) constraint file generated by the Synplify software into the Quartus II software for placement and routing, and for performance evaluation.

5. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

Figure 9–1 shows the recommended design flow when using the Synplify and the Quartus II software.

Figure 9–1. Recommended Design Flow



Notes to Figure 9–1:

(1) The Synopsys Constraints Format (**.scf**) file is generated when you select a Stratix III device. The file contains timing constraints for the TimeQuest timing analyzer.

The Synplify and Synplify Pro software support both VHDL and Verilog HDL source files. The Synplify Pro software also supports mixed synthesis, allowing a combination of VHDL and Verilog HDL source files.

Specify timing constraints and attributes for the design in a Synplify Constraints File (**.sdc**) with the SCOPE window in the Synplify software or directly in the HDL source file. Compiler directives can also be defined in the HDL source file. Many of these constraints are forward-annotated in the Tcl file for Quartus II software use. If a Stratix III device is selected, the Synplify software generates an additional **.scf** file that contains timing constraints for the TimeQuest timing analyzer.

For more details, refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 9–20.

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent register transfer level (RTL) view netlist (**.srs**) and technology-view netlist (**.srm**) files. You can use the Synplify HDL Analyst to visually analyze and debug your design. The HDL Analyst supports cross probing between the RTL and Technology views, the HDL source code, and the Finite State Machine (FSM) viewer. Refer to "FSM Compiler" on page 9–10.

A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro software includes the HDL Analyst.

Once synthesis is complete, import the electronic design interchange format (EDIF) or Verilog Quartus Mapping (VQM) netlist to the Quartus II software for place-and-route. You can use the Tcl file generated by the Synplify software to forward-annotate your constraints, and optionally to set up your project in the Quartus II software. If a Stratix III device is selected, the Quartus II software uses the SDC-format timing constraints from the **.scf** file with the TimeQuest timing analyzer by default. For other devices, the Quartus II software uses the Tcl classic timing analyzer timing constraints written to the Quartus Setting File (**.qsf**).

Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 9–20 for information on how to manually change from the TimeQuest timing analyzer to the classic timing analyzer for Stratix III devices.

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 9–1, 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. Repeat the process until the area and timing requirements are met.

While you can perform simulation at various points in the process, final timing analysis should be performed after placement and routing is complete. Formal verification may also be performed at various stages of the design process.

For more information about how the Synplify software supports formal verification, refer to the *Formal Verification* section 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 & Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

In some cases, you may be required to modify the source code if 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 9–1 lists these file types.

Table 9–1. Synplify Intermediate & Output Files (Part 1 of 2)

File Extensions	File Description
.srs	Technology-independent RTL netlist that can be read only by the Synplify software
.srm	Technology view netlist
.srr *(1)*	Synthesis Report file
.edf/.vqm *(2)*	Technology-specific netlist in electronic design interchange format (EDIF) or VQM file format
.acf/.tcl *(3)*	Forward-annotated constraints file containing constraints and assignments

Table 9–1. Synplify Intermediate & Output Files (Part 2 of 2)

File Extensions	File Description
.scf *(4)*	Synopsys Constraint Format file containing timing constraints for TimeQuest

Notes to Table 9–1:

(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, because it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics that may 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 the logic cell utilization below the Synplify software estimate. Use the device utilization reported by the Quartus II software after place-and-route.

(2) An EDIF output file (**.edf**) is created only for ACEX® 1K, FLEX® 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX® 7000, MAX 9000, and MAX 3000 devices. A VQM file is created for all other Altera device families.

(3) An Assignment and Configuration File (**.acf**) file is created only for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. The ACF is generated for backward compatibility with the MAX+PLUS® II software. 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, if applicable, the MAX+PLUS® II assignments are imported from the ACF file.

(4) This file is created only if a Stratix III device is selected in the Synplify software. For these devices, Synplify software forward annotates the timing constraints in the **.scf** file, and the Tcl file contains the command to use the TimeQuest timing analyzer in the Quartus II software.

Output Netlist File Name & Result Format

Specify the output netlist directory location and name by performing the following steps:

1. On the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab.
3. In the **Results Directory** box, type your output netlist file directory location.
4. In the **Result File Name** box, type your output netlist file name.

By default, directory and file name are set to the project implementation directory and the top-level design module or entity name.

The **Result Format** and **Quartus version** options are also available on the **Implementation Results** tab. The **Result Format** list specifies an EDIF or VQM netlist depending on your device family. The software creates an

EDIF output netlist file only for ACEX 1K, FLEX 10K, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. For other Altera devices, the software generates a VQM-formatted netlist.

Beginning with the Synplify software version 8.4, 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.

Synplify Optimization Strategies

As designs become more complex and require increased performance, using different optimization strategies has become important. Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus II software options can help you obtain the required results.

For additional design and optimization techniques, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 and the *Area & Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

The Synplify software offers many constraints and optimization techniques to improve your design's performance. The Synplify Pro software adds some additional techniques that are not supported in the basic Synplify software. Wherever this document describes Synplify support, this includes both the basic Synplify and the Synplify Pro software; Synplify Pro-only features are labeled as such. This section provides an overview of some of the techniques you can use to help improve the quality of your results.

For more information about applying the attributes discussed in this section, refer to the *Tasks & Tips* chapter of the *Synplify Software User Guide*.

Implementations in Synplify Pro

In the Synplify Pro software, on the Project menu, click **New Implementation** to create different synthesis results without overwriting the others. 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/EDIF, **.scf** (if a Stratix III device is selected) 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 through user-assigned timing constraints to optimize the performance of the design. The Synplify software optimizes the design to attempt to meet these constraints.

The Quartus II NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus II software using either a Tcl script file or a **.scf** file for timing-driven place and route. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 9–20 or "Passing Constraints to the Quartus II Software using Tcl Commands" on page 9–22 for more details about how constraints such as clock frequencies, false paths, and multicycle paths are forward-annotated. This section explains some of the important timing constraints in the Synplify software.

The Synplify Synthesis Report File (**.srr**) contains timing reports of estimated place-and-route delays. The Quartus II software can perform further optimizations on a post-synthesis netlist from third-party synthesis tools. In addition, designs may contain black boxes or 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. The statistics in these reports should be used 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/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, and a t_{SU} value can be inferred by setting an external input delay. The following equations illustrate the relationship between t_{CO}/t_{SU} and the input/output delays:

(1) t_{CO} = clock period – external output delay

(2) t_{SU} = clock period – 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. It is important to specify which paths are multicycle

to avoid having the Quartus II and the Synplify compilers work excessively on a non-critical path. Not specifying these paths can also result in an inaccurate critical path being reported during timing analysis.

False Paths

False paths are paths that should not be considered during timing analysis or which should be assigned low (or no) priority during optimization. Some examples of false paths are slow asynchronous resets and test logic added to the design. Set these paths in the **False Paths** tab of the SCOPE window. Use the `define_false_path` attribute.

FSM Compiler

If the FSM Compiler is turned on, the compiler automatically detects state machines in a design. The compiler can then extract and optimize the state machine. The FSM Compiler analyzes the state machine and decides to implement sequential, gray, or one-hot encoding based on the number of states. It also performs unused-state analysis, optimization of unreachable states, and minimization of transition logic.

If the FSM Compiler is turned off, the compiler does not infer state machines. The state machines are implemented as coded in the HDL code. Thus, if the coding style for the state machine was sequential, then the implementation is also sequential. If the FSM Compiler is turned on, the compiler infers the state machines. The implementation is based on the number of states regardless of the coding style in the HDL code.

You can 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 9–2.

Table 9–2. syn_encoding Directive Values (Part 1 of 2)

Value	Description
Sequential	Generates state machines with the fewest possible flip-flops. 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 flip-flop changes during each transition. Gray-encoded state machines tend to be free of glitches.

Table 9–2. syn_encoding Directive Values (Part 2 of 2)

Value	Description
One-hot	Generates state machines containing one flip-flop 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 binary implementations.
Safe	Generates extra control logic to force the state machine to the reset state if an invalid state is reached. The safe value can be used 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 9–1 shows sample VHDL code for applying the `syn_encoding` directive.

Example 9–1. 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

The Synplify Pro software can use the FSM Explorer to automatically explore different encoding styles for a state machine, and then implement the best encoding based on the overall design constraints. The FSM Explorer uses the FSM Compiler to identify and extract state machines from a design. However, unlike the FSM Compiler which 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 & Options

The following sections list other attributes and options that you can modify in the Synplify software to improve your design performance.

Retiming in Synplify Pro

The Synplify Pro 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-outs, you can 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 the 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, and the minimum allowed value of the attribute is 4.) Using this attribute may result in increased logic resource utilization, thus putting a strain on routing resources and leading to long compile times and difficult fitting.

If you need to duplicate an output register or output enable register, you can create a register for each output pin by using the `syn_useioff` attribute (refer to "Register Packing").

Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets may not be maintained in order 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, it is possible to control register packing with the `syn_useioff` attribute. The `syn_useioff` attribute takes a Boolean value and can be applied to ports or entire modules. Setting the value to **1** instructs the compiler to pack the register into an I/O cell. Setting the value to **0** prevents register packing in both the Synplify and Quartus II software.

Resource Sharing

The Synplify software uses resource sharing techniques during synthesis by default to reduce area. Turning off the **Resource Sharing** option on the **Options** tab of the **Implementation Options** dialog box can improve performance results for some designs. If you turn off this option, be sure to check the results to determine if it helps the timing performance, and if it does not help, then you should leave it 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 and/or architectures. Setting the value to **hard** maintains the boundaries of a module and/or architecture, and prevents cross-boundary optimization.

By default, the Synplify software generates a hierarchical VQM file. To flatten the file, set the `syn_netlist_hierarchy` attribute equal to "0".

Register Input & 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/out of registers to further constrain critical paths.

These options are useful in closing 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 this option, specifying the actual routing delay (from place-and-route results) so that the tool can meet the required clock frequency.

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, you can 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 9–2. 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 syntax for Verilog HDL is shown below:

```
object /* synthesis syn_direct_enable = 1 */ ;
```

Standard I/O Pad

For certain Altera devices and the equivalent device I/O standard, you can specify the I/O standard type to use for the I/O pad in the design using the **I/O Standard** panel in the Synplify SCOPE window.

Example 9–3 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 9–3. 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 *Synplify Reference Manual*.

Altera-Specific Attributes

The following attributes are for use with specific Altera device features. These attributes are forward-annotated to the Quartus II project and are used during the place-and-route process.

altera_chip_pin_lc

Use this attribute to make pin assignments. This attribute takes a string value and applies it to inputs and outputs.

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 9–4 shows VHDL code for making location assignments to ACEX 1K and FLEX 10KE devices.

The "@" is used to specify pin locations for ACEX 1K and FLEX 10KE devices. For these devices the pin location assignments are written to the output EDIF.

Example 9–4. Making Location Assignments to ACEX 1K & FLEX 10KE Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "@14, @5,@16, @15";
```

Example 9–5 shows VHDL code for making location assignments for other Altera devices. The pin location assignments for these devices are written to the output Tcl script.

Example 9–5. Making Location Assignments to Other Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "14, 5, 16,
   15";
```

The `data_out` signal is a 4-bit signal; `data_out[3]` is assigned to pin 14 and `data_out[0]` is assigned to pin 15.

altera_implement_in_esb or altera_implement_in_eab

You can 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® family of devices is not supported by this option. This attribute is ignored for designs targeting devices that do not have ESBs or EABs.)

altera_io_powerup

You can 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.

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 EDIF) file, a **.scf** file (if a Stratix III device is selected), 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 standalone application. Once 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.

You can use NativeLink integration to integrate the Synplify software and Quartus II software with a single GUI for both the 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 on how constraints are passed to the Quartus II software. This section describes the following topics:

Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, you must first verify that the `QUARTUS_ROOTDIR` environment variable contains the Quartus II software installation directory. This environment variable is required to use the Synplify and Quartus II software together.

Under each Implementation in the Synplify Pro software, you can create a place-and-route implementation called **pr_**<*number*> **(Altera Place & Route)**. You can create new place and route implementations using the **New P&R** button in the GUI. To run the Quartus II software in command-line mode after each synthesis run, use the text box to turn on the place-and-route implementation. The results of the place and route are written to a log file in the **pr_**<*number*> directory under the current implementation directory.

You can also use the commands in the Quartus II menu to run the Quartus II software at any time following a successful completion of synthesis. Use one of the following commands from the **Quartus II** submenu under the Options menu in the Synplify software:

- **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. You can use this to configure options for the project and execute any Quartus II commands.
- **Run Background Compile**—Runs the Quartus II software in command-line mode with the project settings from the synthesis run. The results of the place-and-route are written to a log file.

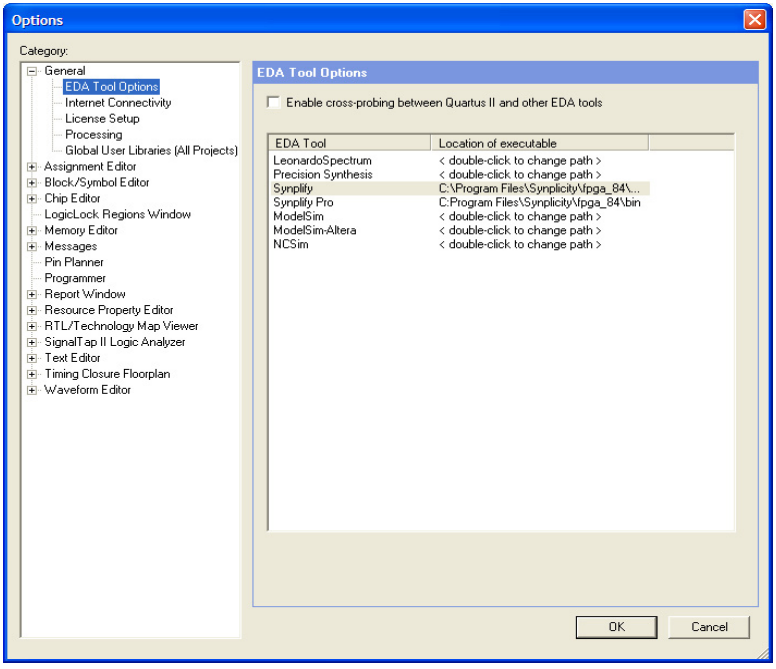
The <*project_name*>**_cons.tcl** file is used to set up the Quartus II project and calls the <*project_name*>**.tcl** file to pass constraints from the Synplify software to the Quartus II software. By default, the <*project_name*>**.tcl** file contains device, timing, and location assignments. If a Stratix III device is is selected, the <*project_name*>**.tcl** file contains the command to use the Synplify-generated **.scf** constraints file with the TimeQuest timing analyzer instead of using the Tcl constraints with the classic timing analyzer.

Using the Quartus II Software to Launch 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 synthesize a design as part of a normal compilation in the Quartus II software.

To set up Synplify in Quartus II, on the Tools menu, click **Options**. In the **Options** window, click **EDA Tool Options** and specify the path of Synplify or Synplify Pro software, as shown in Figure 9–2.

Figure 9–2. Specifying the Path to the Synplify Software



For detailed information about using NativeLink integration with the Synplify software, refer to the Quartus II Help.

Running the Synplify software with NativeLink integration requires a floating network license (as opposed to a node-locked single-PC license), because batch mode compilation is supported only with floating licenses.

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, perform the following steps:

1. Ensure the VQM/EDIF, **.scf** (if a Stratix III device is selected), and Tcl files are located in the same directory (they should be located in the implementation directory by default).

2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console opens.

3. At the Tcl Console command prompt, type:

```
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 Synplify are passed to Quartus II software to be used by the TimeQuest timing analyzer.

Timing constraints you set in Synplify are stored in the Synplify Design Constraints (**.sdc**) file. The timing constraints are forward-annotated using the Tcl file for most devices, as described in "Passing Constraints to the Quartus II Software using Tcl Commands" on page 9–22. If a Stratix III device is selected, Synplify generates a Synopsys Constraints Format (**.scf**) file and the timing constraints are forward-annotated using the SDC format. By default, this file is used by the TimeQuest timing analyzer in the Quartus II software.

It is recommended that you use the TimeQuest timing analyzer for Stratix III devices. However, you can continue to 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, select **Settings**.

2. Click **Timing Analysis Settings.**

3. Under **Timing analysis processing,** click the **Use Classic Timing Analyzer during compilation** radio button. Click **OK**.

Synplicity recommends you to modify constraints using the SCOPE constraint editor window and not through the generated **.sdc**, **.scf** or Tcl file.

For addition information about TimeQuest, refer to the *TimeQuest Timing Analyzer* chapter in the *Quartus II Handbook*.

Some of the constraints you set in Synplify are not supported by the TimeQuest timing analyzer and are not forward-annotated in the **.scf** 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 9–22, however, the constraints are mapped to SDC commands for the TimeQuest timing analyzer. For the syntax and arguments for these commands, refer to the applicable subsection or refer to the Synplify Help. For a list of corresponding commands in the Quartus II software, refer to the Quartus II Help.

Individual Clocks & Frequencies

You can specify clock frequencies for individuals clocks in Synplify software with the command, `define_clock`. This command is passed to Quartus II software with `create_clock`.

Input & Output Delay

You can specify input delay and output delay constraints in 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

You can specify a multicycle path constraint in Synplify with the command `define_multicycle_path`. This command is passed to the Quartus II software with `define_multicycle_path`.

False Path

You can specify a false path constraint in 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 discusses 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 9–20 for information about timing constraints supported by TimeQuest.

Global Signals

The Synplify software automatically promotes clock signals to global routing lines and passes **Global Signal** assignments to the Quartus II software. The assignments ensure that the same global routing constraints are applied during placement and routing.

Note that the signals promoted to global routing can be different than the ones that the Quartus II software promotes to global routing by default. Synplify 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` Synplify 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 & Frequencies

You can specify clock frequencies for individual clocks with the following Synplify commands:

Example 9–6. Specifying Clock Frequencies for Individual Clocks

```
define_clock -name {<clock_name>} -freq <frequency> -clockgroup <clock_group>
-rise <rise_time> -fall <fall_time>
define_clock -name {<clock_name>} -period <period> -clockgroup <clock_group>
-rise <rise_time> -fall <fall_time>
```

Table 9–3 shows the command arguments.

Table 9–3. Command Arguments

Argument	Description
`-name`	The <*clock_name*> specifies a design port name or a 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`. Synplify 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 9–3. Command Arguments

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 9–3:

(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*> ÷ 10.

(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 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. You can use the corresponding constraints for period.

Virtual Clocks

The Quartus II software supports virtual clocks. If you use the virtual clock setting in Synplify, the setting is mapped to a constraint in the Quartus II software.

Route Delay Option

The `-route` option in Synplify clock constraints is designed for use 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 9–7.

Example 9–7. Specifying Clock Frequencies for Multiple Clocks

```
define_clock -name {<clock_name1>} -freq <frequency1> \
-clockgroup <clock_group1> -rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> \
-clockgroup <clock_group2> -rise <rise_time2> -fall <fall_time2>
```

<*clock_group1*> and <*clock_group2*> are unique names defined in the Synplify software for base clock settings in the Quartus II software.

If the clock <*rise_time*> is zero ("0"), multiple separate clocks are passed to the Quartus II software with the commands shown in Example 9–8:

Example 9–8. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target mapped_clock_name1 <base_clock_setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty_cycle2> \
-target mapped_clock_name2 <base_clock_setting2>
```

If the clock <*rise_time*> is non-zero, multiple separate clocks are passed to the Quartus II software with the following commands shown in Example 9–9:

Example 9–9. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Not Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty cycle1> \
-no_target <base clock setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty cycle2> \
-no_target <base clock setting2>

create_relative_clock -base_clock <base clock setting1> -offset <rise time1>ns \
-duty_cycle <duty cycle1> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name1> <derived clock setting1>

create_relative_clock -base_clock <base clock setting2> -offset <rise time2>ns \
-duty_cycle <duty cycle2> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name2> <derived clock_setting2>
```

Multiple Clocks with Different Frequencies in the Same Clock Group

You can specify multiple clocks with relative clock settings in the same clock group in Synplify with different frequencies with the commands shown in Example 9–10:

Example 9–10. Specifying Multiple Clocks with Different Frequencies in the Same Clock Group

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group1> \
-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 ("0"), 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 9–11:

Example 9–11. Quartus II Assignments for Multiple Clocks with Different Frequencies in the Same Clock Group, if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target <mapped_clock_name1> <base_clock_setting1>

create_relative_clock -base_clock <base_clock_setting1> \
-duty_cycle <duty_cycle2> -multiply <multiply_by> -divide <divide_by> \
-target <mapped_clock_name2> <derived_clock_setting2>
```

Inter-Clock Relationships—Delays & False Paths between Clocks

You can set a clock-to-clock delay constraint in Synplify with the commands in Example 9–12.

Example 9–12. Specifying Clock-to-Clock Delay Constraints

```
define_clock_delay -fall <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -fall <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -fall <clock_name1> -fall <clock_name2> <delay_value>
```

If <*delay_value*> is set to `false`, these constraints in Synplify indicate a false path between the two clocks. If all four rise/fall clock-edge pairs are specified in the Synplify software, the Synplify constraints are mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <clock_name1> \
-to <clock_name2>
```

If all four clock-edge pairs are not specified in Synplify, the constraint cannot be mapped to a constraint for the Quartus II classic timing analyzer.

If <*delay_value*> is set to a value other than `false`, these constraints in Synplify is not mapped to a constraint in the Quartus II software. The Quartus II classic timing analyzer does not support clock-edge to clock-edge delay constraints.

False Paths

You can specify the false path constraint in Synplify with the command shown below.

```
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 Synplify is mapped to the constraint in the Quartus II software, as shown below.

```
set_timing_cut_assignment -from <sig_name1> \
-to <sig_name2>
```

Synplify 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 software with the commands shown below.

```
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 you can expand appropriately 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

You can specify a false path constraint from a signal in Synplify with the following command:

```
define_false_path -from <sig_name>
```

The Quartus II software 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
You can specify a false path constraint to a signal in Synplify with the following command:

```
define_false_path -to <sig_name>
```

The Quartus II software 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 Synplify is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from {*} -to <sig_name>
```

False Path Through a Signal
You can specify a false path constraint through a signal in Synplify with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3>
```

The Quartus II software does not currently support false paths with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II classic timing analyzer.

Multicycle Paths

You can specify a multicycle path constraint in Synplify with the following command:

```
define_multicycle_path -from <sig_name1> \
-to <sig_name2> <clock_cycles>
```

This constraint in Synplify 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 9–10.

<*clock_cycles*> is the number of clock cycles for the multicycle path.

Multicycle Path from a Signal
You can specify a multicycle path constraint from a signal in Synplify 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 software, similar to the false path constraints:

```
set_multicycle_assignment -from <sig_name> \
-to {*} <clock_cycles>
```

Multicycle Path to a Signal
You can specify a multicycle path constraint to a signal in Synplify 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 software, similar to the false path constraints:

```
set_multicycle_assignment -from {*} <sig_name> \
<clock_cycles>
```

Multicycle Path Through a Signal
You can specify a multicycle path constraint through a signal in Synplify using the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3> <clock_cycles>
```

The Quartus II software does not currently support multicycle paths with a **through path** specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II classic timing analyzer.

Maximum Path Delays

You can specify the maximum path delay relationships between signals in Synplify with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \
-max <delay_value>
```

This constraint in Synplify 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 software does not support signal groups or bus notation, and supports only register names for this constraint.

Maximum Path Delay from a Signal
You can specify the maximum path delay constraint from a signal in Synplify 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 software, similar to false path constraints:

```
set_instance_assignment -from <sig_name> -to {*} \
-name SETUP_RELATIONSHIP <delay_value>ns
```

Maximum Path Delay to a Signal
You can specify the maximum path delay constraint to a signal in Synplify 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 software, 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
You can specify the maximum path delay constraint through a signal in Synplify with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \
-through <sig_name3> -max <delay_value>
```

The Quartus II classic timing analyzer does not support maximum path delay constraints with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II classic timing analyzer.

Register Input & Output Delays
These register input delay and register output delay constraints in Synplify are for use in synthesis only, and therefore are not forward-annotated to the Quartus II software.

Default External Input Delay
You can specify the default input delay constraint in Synplify 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
You can specify a port-specific input delay constraint in Synplify 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
You can specify the default output delay constraint in Synplify 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
You can specify a port-specific input delay constraint in Synplify 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 & Architecture-Specific Features

Altera provides parameterizable megafunctions including the LPMs, device-specific Altera megafunctions, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP℠). You can use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code.

If you want to instantiate a megafunction in your HDL code, you can do so by using the MegaWizard Plug-In Manager to parameterize the function or instantiating the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface within the Quartus II software for customizing and parameterizing any available megafunction for the design. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 9–34 describes the MegaWizard Plug-In Manager flow with the Synplify software.

For more information about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, refer to the appropriate IP documentation.

The 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 9–39.

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*. The *Recommended HDL Coding Styles* chapter also provides details on using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard, as well as providing coding style recommendations and HDL examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

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 that instantiates the megafunction (a black box methodology). Some megafunctions also support the generation of a fully synthesizeable netlist for improved results with EDA synthesis tools such as Synplify (a clear box methodology). Clear- and black-box methodologies are described in the following sections.

Clear Box Methodology

You can use the MegaWizard Plug-In Manager to generate a fully synthesizeable netlist. This flow is referred to as a clear box methodology because the Synplify software can "see" into the megafunction file. The clear box feature enables the synthesis tool to report more accurate timing estimates and resource utilization, and to take better advantage of timing-driven optimization than a black box methodology.

For certain megafunctions, the clear box feature is enabled by turning the **Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** option on in the MegaWizard Plug-In Manager. If the option does not appear, then clear box models are not supported for the selected megafunction. The Synplify software supports clear box models for the Stratix and Cyclone® series devices only. Turning this option on causes the Quartus II MegaWizard Plug-In Manager to generate a synthesizeable clear box netlist instead of the megafunction wrapper file described in "Black Box Methodology" on page 9–35.

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Clear Box 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. This file can help you instantiate the megafunction clear box netlist file, <*output file*>**.v**, in your top-level design. Include the megafunction clear-box netlist file in your Synplify Project. Finally, include the megafunction clear box netlist file, <*output file*>**.v**, along with your Synplify-generated VQM netlist in your Quartus II project.

Using MegaWizard Plug-In Manager-Generated VHDL Files for Clear Box Megafunction Instantiation

If you check the <*output file*>**.cmp** and <*output file*>**_inst.vhd** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL Component declaration file and a VHDL Instantiation template file for use in your design. These files help to instantiate the megafunction clear box netlist file, <*output file*>**.vhd**, in your top-level design. Include the megafunction clear box netlist file in your Synplify Project. Finally, include the megafunction clear box netlist file, <*output file*>**.vhd**, along with your Synplify-generated VQM netlist in your Quartus II project.

Black Box Methodology

Using the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black-box methodology because the megafunction is treated as a black box in the Synplify software. The black box methodology does not allow the synthesis tool any visibility into the function module and therefore 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. Refer to "Other Synplify Software Attributes for Creating Black Boxes" on page 9–38 for details.

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Black Box Megafunction Instantiation

If you check the <*output file*>**_inst.v** and <*output file*>**_bb.v** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file and a hollow-body black-box module declaration for use in your Synplify design. The instantiation template file helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level design. Do not include the megafunction variation wrapper file in your Synplify Project, but add it, with your Synplify-generated VQM netlist, to your Quartus II project. Add the hollow-body black-box module declaration <*output file*>**_bb.v** to your Synplify project to describe the port connections of the black box.

The Synplify software reads black box instantiations for the `alt_pll` megafunction and writes the phase-locked loop (PLL) instance into the resulting VQM output netlist. Reading the PLL function allows the Synplify software to interpret the multiplication and division factors in the PLL instantiation to make the correct timing assignments. Therefore, for `alt_pll` instantiations, make sure to include the megafunction variation wrapper file <*output file*>**.v** in your Synplify project and do not declare it as a black box. Because the instance is written in the VQM file, do not include the `alt_pll` megafunction variation wrapper file <*output file*>**.v** in your Quartus II project.

You can use the `syn_black_box` compiler directive to declare a module as a black box. The top-level design files must contain the megafunction port mapping and hollow-body module declaration, as described above. You can apply the `syn_black_box` directive to the module declaration in the top-level file or a separate file included in the project (such as the <*output file*>**_bb.v** file) 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 9–38.

Example 9–13 shows a sample top-level file that instantiates **verilogCount.v**, which is a customized variation of the `lpm_counter` generated by the MegaWizard Plug-In Manager.

Example 9–13. Top-Level Verilog HDL Code with Black Box Instantiation of lpm_counter

```
module topCounter (clk, count);
    input clk;
    output[7:0]  count;
    verilogCounter verilogCounter_inst (
        .clock ( clk ),
        .q ( count )
);
endmodule
// Module declaration found in verilogCounter_bb.v
// The following attribute is added to create a
// black box for this module.
module verilogCounter (
    clock,
    q) /* synthesis syn_black_box */;
    input clock;
    output[7:0]  q;
endmodule
```

Using MegaWizard Plug-In Manager-Generated VHDL Files for Black Box Megafunction Instantiation

If you check the <*output file*>**.cmp** and <*output file*>**_inst.vhd** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL component declaration file and a VHDL instantiation template file for use in your Synplify design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>**.vhd**, in your top-level design. Do not include the megafunction variation wrapper file in your Synplify project, but add it, along with your Synplify-generated VQM netlist, to your Quartus II project.

The Synplify software reads black box instantiations for the `alt_pll` megafunction and writes the phase-locked loop (PLL) instance into the resulting VQM output netlist. Reading the PLL function allows the Synplify software to interpret the multiplication and division factors in the PLL instantiation to make the correct timing assignments. Therefore, for `alt_pll` instantiations, make sure to include the megafunction variation wrapper file <*output file*>**.vhd** in your Synplify project and do not declare it as a black box. Because the instance is written in the VQM file, do not include the `alt_pll` megafunction variation wrapper file <*output file*>**.vhd** in your Quartus II project.

You can 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, as described above. 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 9–14 shows a sample top-level file that instantiates **vhdlCount.vhd**, which is a customized variation of the `lpm_counter` generated by the MegaWizard Plug-In Manager.

Example 9–14. Top-Level VHDL Code with Black Box Instantiation of lpm_counter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY testCounter IS
    PORT
    (
        clk: IN STD_LOGIC ;
        count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
    END testCounter;
ARCHITECTURE top OF testCounter IS
component vhdlCount
    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 vhdlCount: component is true;
BEGIN
    vhdlCount_inst : vhdlCount PORT MAP (
        clock => clk,
        q => count
    );
END top;
```

Other Synplify Software Attributes for Creating Black Boxes

The black box methodology does not allow 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 9–15 for a Verilog HDL example.

Example 9–15. Verilog HDL Example

```
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 *Tasks & Tips* chapter of the *Synplify User Guide*.

Inferring Altera Megafunctions from HDL Code

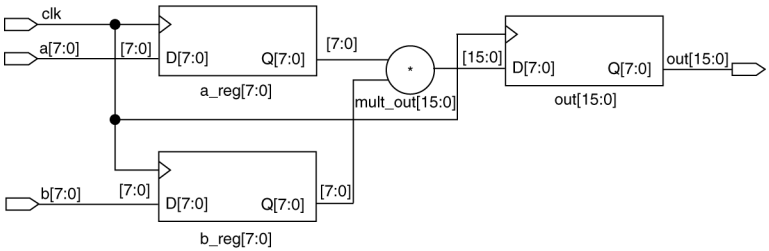
The Synplify software uses Behavior Extraction Synthesis Technology (BEST) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, and so forth. It then keeps the structures abstract for as long as possible in the synthesis process. This allows the use of technology-specific resources 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 9–3 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 `lpm_mult` megafunction. For devices with DSP blocks, the software may implement the `lpm_mult` function in a DSP block instead of regular logic, depending on device utilization. For Stratix II and Stratix III devices, the software maps directly to DSP block device atoms instead of instantiating a megafunction in the **.vqm** file.

Figure 9–3. 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 may or may not be in the critical paths, are mapped to DSP blocks. Smaller multipliers and multipliers that are not in the critical paths may 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

Multipliers can be implemented 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 below:

<*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 9–4 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 9–4. 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 atoms (for supported devices)

Example 9–16 and Example 9–17 show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

Example 9–16. 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 9–17. 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 RAM architecture. For Stratix II and Stratix III devices, the software maps directly to RAM block device atoms instead of instantiating a megafunction in the VQM file.

Follow the guidelines below 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 may not be mapped to RAM blocks, so avoid blocking statements when modeling RAMs in Verilog HDL.

For certain device families, the `syn_ramstyle` attribute specifies the implementation to use for an inferred RAM. You can apply `syn_ramstyle` globally, to a module, or to a RAM instance, to specify `registers` or `block_ram` values. To turn off RAM inference, set the attribute value to `registers`.

When inferring RAM for certain Altera device families, the Synplify software generates additional bypass logic. This logic is generated to resolve a half-cycle read/write behavior difference between the RTL and post-synthesis simulations. The RTL simulation shows the memory being updated on the positive edge of the clock, and the post-synthesis simulation shows the memory being updated on the negative edge. To eliminate the 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 the bypass logic.

For the Stratix and Cyclone series devices, you can 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 9–18 shows sample VHDL code for inferring dual-port RAM.

Example 9–18. 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 9–19 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 9–19. 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

You can use Verilog 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 9–20 and Example 9–21 illustrate how RAM memories can be initialized through HDL code, and how the corresponding HEX file is generated using Verilog HDL.

Example 9–20. 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)
       begin
          mem[waddr] <= data;
       end
end
```

Example 9–21. Sample VQM Instance Containing Memory Initialization File from Example 9–20

```
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 RAM block is inferred from an HDL design, the software uses an Altera megafunction to target the device RAM architecture. For Stratix II and Stratix III devices, the software maps directly to RAM block device atoms instead of instantiating a megafunction in the **.vqm** file. Follow the guidelines below for the Synplify software to successfully infer ROM in a design:

- The address line must be at least two-bits wide.
- 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 it is 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 can improve the design performance.

Incremental Compilation & 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 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 a particular design partitions and merging results with previous compilation results of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual subblocks and then preserve the results before you integrate the blocks into a final design and optimize it at the top level.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro 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 Quartus II incremental compilation and LogicLock™ design methodologies. 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 (basic Synplify and Synplify Pro) by creating a separate project for the logic in each partition of the design. Creating different netlist files for each partition of the design means that each partition is independent of the others. When synthesizing the entire project, only portions of a design that have been updated have to be resynthesized when you compile the design. You can make changes and resynthesize one partition of a design to create a new netlist file without affecting the synthesis results and placement of other partitions.

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 the incremental compilation and methodologies 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. Following the guidelines in this section can help you achieve good results with these methodologies.

The following list shows the general top-down compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files as in the regular design flow.
2. Determine which hierarchical blocks are to be treated as separate partitions in your design.
3. Set up your design using the MultiPoint feature or separate projects so that a separate netlist file is created for each partition of the design.
4. If using separate projects, disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and technology-map each partition in the Synplify Pro or Synplify software, making constraints as you would in the regular design flow.
6. Import the VQM or EDIF netlist and the 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 & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about using the LogicLock feature in the Quartus II software, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

Hierarchy & Design Considerations with Multiple VQM Files

To ensure the proper functioning of the synthesis flow, you can create separate netlist files only for modules and entities. In addition, each module or entity should 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 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. 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 *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design with Separate Netlist Files

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 that you can take advantage of the incremental compilation and LogicLock design flows 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 generate multiple VQM files either by using the MultiPoint synthesis flow and LogicLock attributes in the Synplify Pro 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 only), you create multiple VQMs from one easy-to-manage top-level synthesis project. By using the manual black box method (Synplify or Synplify Pro), you have multiple synthesis projects, which may be required for certain team-based or bottom-up designs where a single top-level project is not desired.

Once you have created multiple VQM files using one of these two methods, you need to create the appropriate Quartus II projects to place-and-route the design.

Creating a Design with Multiple VQM Files Using Synplify Pro MultiPoint Synthesis

This section describes how to generate multiple VQM files using the Synplify Pro MultiPoint synthesis flow. You must first set up your compile points, constraint files, and Synplify Pro options, then apply Altera-specific attributes to write multiple VQM files and create LogicLock region assignments.

Set Compile Points & 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 Synplify Pro LogicLock constraints to a Compile Point module, then a separate netlist is created for that Compile Point, isolating that logic from any other logic in the design.

Figure 9–6 on page 9–57 shows an example of a design hierarchy that can be split into multiple partitions. The top-level block of each partition can be synthesized as a separate Compile Point.

In this case, modules A, B, and F are Compile Points. The top-level Compile Point consists of the top-level block in the design (that is, block-A in this example), including the logic that is not defined under another Compile Point. In this example, the design for top-level Compile Point A also includes the logic in one of its subblocks, C. Because block F is defined as its own Compile Point, it is not treated as part of the top-level Compile Point A. Another separate Compile Point B contains the logic in blocks B, D, and E. One netlist is created for the top-level module A and submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

Apply Compile Points to the module or architecture in the Synplify Pro SCOPE spreadsheet or in the **.sdc** file. You cannot set a Compile Point in the Verilog HDL or VHDL source code. You can set the constraints manually using Tcl or by editing the **.sdc** file. You can also use the GUI which provides two methods, manual or automated, as shown below.

Defining Compile Points Using Tcl or SDC

To set Compile Points using Tcl or an **.sdc** file, use the `define_compile_point` command, as shown in Example 9–22.

Example 9–22. The define_compile_point Command

```
define_compile_point [-disable] [-comment <comment>] <objname> \
[-type <compile point type>]
```

In the syntax statement above, *objname* represents any module in the design. Currently, `locked` is the only Compile Point type supported.

Each Compile Point has a set of constraint files that begin with the `define_current_design` command to set up the SCOPE environment, as shown below.

```
define_current_design {<my_module>}
```

Manually Defining Compile Points from the GUI

The manual method requires you to separately create constraint files for the top-level and the lower-level Compile Points. To use the manual method:

1. From the top level, select the **Compile Points** tab in the SCOPE spreadsheet.

2. Select the modules that you want to define as Compile Points.

   Currently, locked Compile Points are the only type supported. All Compile Points must be defined from the top level because the **Compile Points** tab is not available in the SCOPE spreadsheet from lower level modules.

3. Manually create a constraint file for each module.

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option on the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

Automatically Defining Compile Points from the GUI

When you use the automated process, the lower-level constraint file is created automatically. This eliminates the manual step necessary to set up each Compile Point. To use the automated method, perform the following steps:

1. On the File menu, select **New**. Click to create a new **Constraint File**, or click the **SCOPE** icon in the tool bar.

2. From the **Select File Type** tab of the **Create a New SCOPE File** dialog box, select **Compile Point**.

3. Select the module you want to designate as a Compile Point. The software automatically sets the Compile Points in the top-level constraint file and creates a lower-level constraint file for each Compile Point.

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option on the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

When using Compile Points with the incremental compilation or LogicLock design flow, 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 Pro 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 Pro software.

For more information about Compile Points, refer to the *Synplify Pro User Guide and Reference Manual* on the Synplicity web site at **www.synplicity.com/literature/index.html**.

Apply the LogicLock Attributes

To instruct the Synplify Pro software to create a separate VQM netlist file for each Compile Point, you must indicate that the Compile Point is being used with LogicLock regions in the incremental compilation or LogicLock design methodology. Since separate netlist files are required for incremental compilation, you must use the LogicLock attributes if you plan to use the incremental compilation feature in the Quartus II software. When you apply the appropriate LogicLock attributes, the Synplify Pro software also writes out Tcl commands for the Quartus II software to create a LogicLock region for each netlist.

LogicLock regions in the Quartus II software have size and location properties. The region's size is defined by the height and width of the rectangular area. If the region is specified as auto-size, then the Quartus II software determines the appropriate size to fit the logic assigned to the region. When you specify the size, you must include enough device resources to accommodate the assigned logic. The location of a region is defined by its origin, which is the position of its bottom-left corner or top-left corner, depending on the target device family. In the Quartus II

software, this location can be specified as locked or floating. If the location is floating, the Quartus II software determines the location during its optimization process.

Floating locations are the only type currently supported in the Synplify Pro software.

When you use incremental compilation in the Quartus II software, you should lock down the size and location of the region in the Quartus II software after the first compilation to achieve the best quality of results.

Table 9–5 shows the valid combinations of the LogicLock attributes.

Table 9–5. LogicLock Location & Size Properties

altera_logiclock_location Attribute	altera_logiclock_size Attribute	Description
Floating	Auto	The most flexible type of LogicLock constraint. Allows the Quartus II software to choose the appropriate region size and location.
Floating	Fixed	Assumes the size of LogicLock constraint area is already optimal in the existing Quartus II project.

You can apply these attributes to the top-level constraint file or to the individual constraint files for each lower-level Compile Point. You can use the **Attribute** tab of the SCOPE spreadsheet to set attributes.

Synplify Pro offers another attribute, `syn_allowed_resources`, which restricts the number of resources for a given module. You can apply the `syn_allowed_resources` attribute to any Compile Point view.

For specific information regarding these attributes, refer to the Synplify Pro online help or reference manual.

Creating a Quartus II Project for Multiple VQM Files

During compilation, the Synplify Pro software creates a <*top-level project*>**.tcl** file that provides the Quartus II software with the appropriate constraints and LogicLock assignments, creating a region for each VQM file along with the information to set up a Quartus II project.

The Tcl file contains the following commands for each LogicLock region. Example 9–23 is for module `A` (instance `u1`) in the project named `top` where the region name `cpll_1` was selected by Synplify Pro for the Compile Point.

Example 9–23. Commands for Each LogicLock Region in a Tcl File

```
set_global_assignment -section_id{taps_region} -name{LL_AUTO_SIZE}{ON}
set_global_assignment -section_id{taps_region} -name{LL_STATE}{FLOATING}
set_instance_assignment -section_id{taps_region} -to{|taps:u1} \
-name{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*.

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.

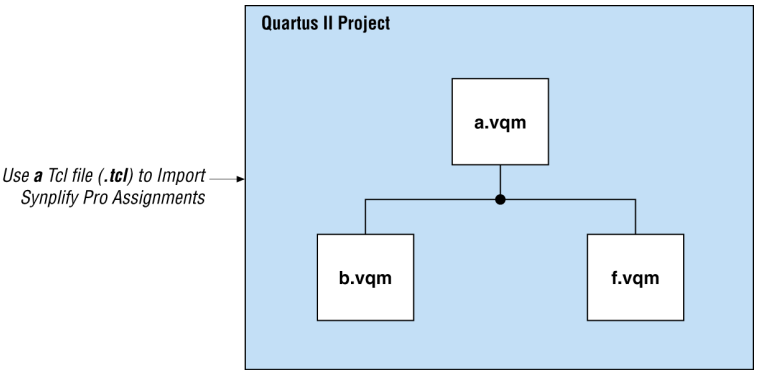
You may require a bottom-up design flow if each partition must be optimized separately, such as in certain team-based design flows. To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental 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 that contains the Synplify Pro assignments for all partitions within the project. This method allows you to import all the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation-time reduction incremental compilation offers. Figure 9–4 shows a visual representation of the design flow for the example design in Figure 9–6.

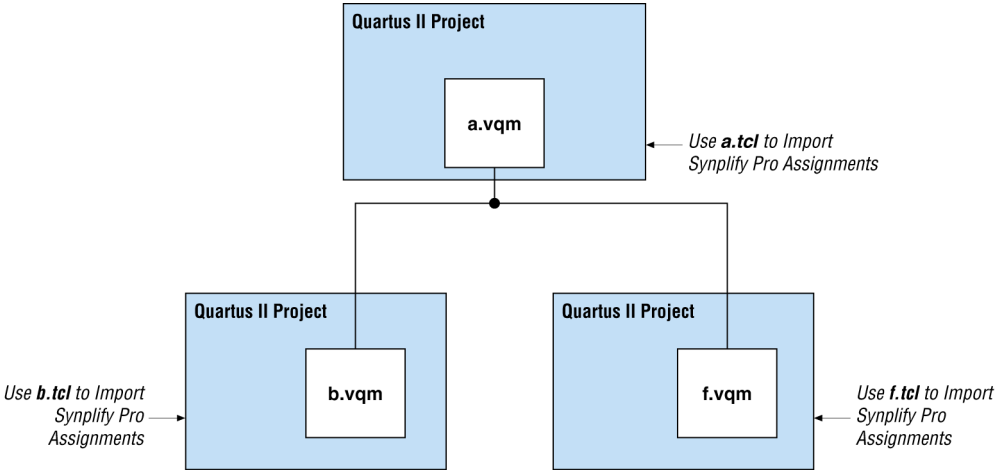
Figure 9–4. Design Flow Using Multiple VQM Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up LogicLock Design Flow

Generate multiple Quartus II projects, one for each partition and netlist in the design. Each designer in the project can optimize their partition separately within the Quartus II software and export the placement for their partitions. Figure 9–5 shows a visual representation of the design flow for the example design in Figure 9–6. The optimized sub-designs can be brought into one top-level Quartus II project using incremental compilation.

Figure 9–5. Design Flow Using Multiple VQM Files with Multiple Quartus II Projects



Generating a Design with Multiple VQM Files Using Black Boxes

This section describes how to manually generate multiple VQM files using black boxes. This manual flow is supported in versions of the Synplify software that do not include the MultiPoint Synthesis feature.

Manually Creating Multiple VQM Files Using Black Boxes

To create multiple VQM files manually in the Synplify software, create a separate project for each low-level module and the top-level design that you want to maintain as a separate VQM file. 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.

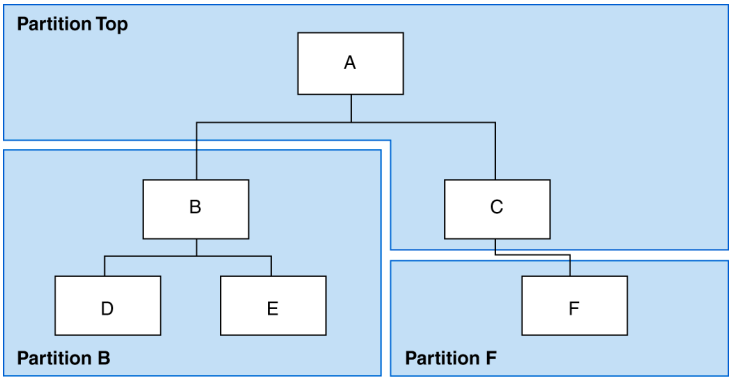
   Modules may include black box instantiations of lower-level modules that are also maintained as separate VQM files.
3. Add constraints with the SCOPE constraint window.
4. Enter the clock frequency to ensure that the sub-design is correctly optimized.
5. In the **Attributes** tab, set **syn_netlist_hierarchy** 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 9–6 shows an example of a design hierarchy that is split up into multiple partitions.

Figure 9–6. Partitions in a Hierarchical Design



In Figure 9–6, the partition top contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in one of its sub-blocks, C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers 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 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 9–24 shows an example of the **A.v** top-level file. Follow the same procedure below for lower-level files which also contain a black box for any module beneath the current level hierarchy.

Example 9–24. Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

    // Any other code in A.v goes here.

endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
boxes.

module B (data_in, clk, ld, data_out) /* synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module F (d, clk, e, q) / *synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project are 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 need a component declaration for the black box just like any other block in the design.

Although VHDL is not case-sensitive, VQM (a subset of Verilog HDL) is case-sensitive. Entity names and their port declarations are forwarded to the VQM. Black box names and port declarations are also passed to the VQM. To prevent case-based mismatches between VQM, use the same capitalization for black box and entity declarations in VHDL designs.

Example 9–25 shows an example of the **A.vhd** top-level file. Follow this same procedure for any lower-level files that contain a black box for any block beneath the current level of hierarchy.

Example 9–25. VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY synplify;
use synplify.attributes.all;

ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
   clk, e, ld : IN STD_LOGIC;
   data_out : OUT INTEGER RANGE 0 TO 15 );
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
   data_in : IN INTEGER RANGE 0 TO 15;
   clk, ld : IN STD_LOGIC;
   d_out : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

COMPONENT F PORT(
   d : IN INTEGER RANGE 0 TO 15;
   clk, e: IN STD_LOGIC;
   q : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

attribute syn_black_box of B: component is true;
attribute syn_black_box of F: component is true;

-- Other component declarations in A.vhd go here
```

```
signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN

U1 : B
PORT MAP (
    data_in => data_in,
    clk => clk,
    ld => ld,
    d_out => cnt_out );

U2 : F
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => data_out );

-- Any other code in A.vhd goes here

END a_arch;
```

After you have completed the steps described in this section, you have a netlist file for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

Creating a Quartus II Project for Multiple VQM Files

The Synplify software creates a Tcl file for each VQM file, that provide the Quartus II software with the appropriate constraints and information to set up a project. For details on 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 9–20.

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 may require a bottom-up design flow where each partition must be optimized separately, such as in certain team-based design flows. To perform a bottom-up compilation in the Quartus II software, create

separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

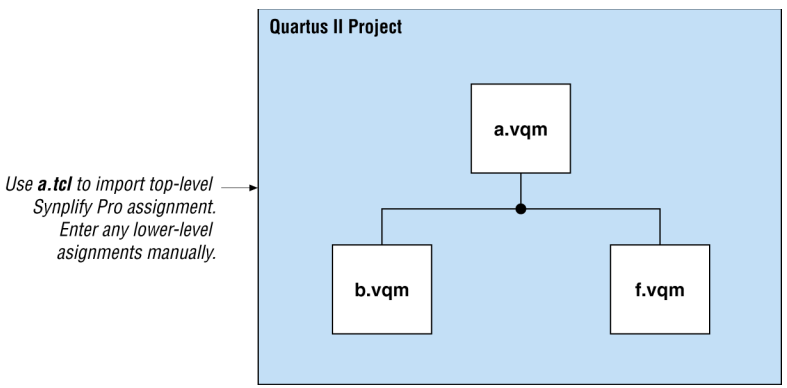
The following sections describe how to create the Quartus II projects for these two design flows.

Creating Compile Points in 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 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 9–7 shows a visual representation of the design flow for the example design in Figure 9–6.

All the constraints from the top-level project will be passed to the Quartus II software in the top-level Tcl file, but any constraints made in the lower-level projects within the Synplify software is not forward-annotated. Enter these constraints manually in your Quartus II project.

Figure 9–7. Design Flow Using Multiple VQM Files with One Quartus II Project

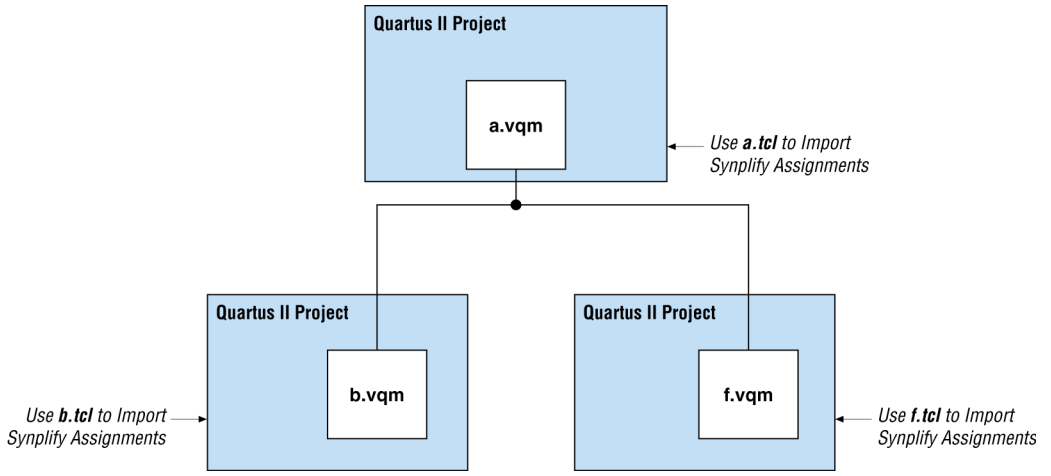


Creating Multiple Quartus II Projects for a Bottom-Up Design Flow

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. Each designer in the project can optimize their block separately within the Quartus II software and export the placement of their blocks. Figure 9–8 shows a visual representation of the design flow for the example in Figure 9–6 on page 9–57. 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. This method allows each block in the design to be treated separately; each block can then be imported into one top-level project.

Figure 9–8. Design Flow Using Multiple Synplify Projects & Multiple Quartus II Projects



Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Synplicity Synplify and 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. Several of the methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Document Revision History

The following table lists the revision history for this chapter:

Date / Version	Changes Made	Summary of Changes
November 2006 v6.1.0	• Chapter 9 was formally Chapter 8 in version 6.0.0. • Added that SCF is generated to pass SDC constraints for TimeQuest. • Added timing constraint information when using TimeQuest. • Moved note about alt_pll megafunctions from clear box section to black box section. • Clarified that Synplify reads the alt_pll megafunction black box file for Stratix and Cyclone series devices.	Updated to include Stratix III support and added information on how to pass timing constraint information for TimeQuest.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Updated cross probing information. • Added NativeLink® integration information. • Added Synplify design flow support. • Added Altera megafunction guidelines and architecture-specific features.	
December 2005 v5.1.1	Minor typographic update.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 8 was formerly chapter 9 in version 5.0.	
May 2005 v5.0.0	Chapter 9 was formerly chapter 7 in version 4.2.	
December 2004 v2.1.0	• Chapter 8 was formerly Chapter 9 in version 4.1. • Updated information. • New functionality for Quartus II software version 4.2. • Updated figure 8-1.	
June 2004 v2.0.0	• Updates to tables, figures. • New functionality for Quartus II software version 4.1.	
February 2004 v1.0.0	Initial release.	

10. Mentor Graphics LeonardoSpectrum Support

QII51010-6.1.0

Introduction

As programmable logic devices (PLDs) become more complex and require increased performance, advanced synthesis has become an important part of the design flow. Combining HDL coding techniques, Mentor Graphics LeonardoSpectrum™ software constraints, and Quartus® II options provide the performance increase needed for today's system-on-a-programmable-chip (SOPC) designs.

The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices. The LeonardoSpectrum software version 2005b supports the Stratix® II, Stratix, Stratix GX, Cyclone™ II, Cyclone, MAX® II, MAX series, APEX™ series, FLEX® series, and ACEX® series device families. Altera® recommends using the advanced Precision Synthesis software for new designs in new device families.

For more information about Precision RTL Synthesis, refer to the *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*.

This chapter documents key design methodologies and techniques for achieving better performance in Altera devices using the LeonardoSpectrum and Quartus II design flow.

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 web site at **www.mentor.com**. For information about installing the LeonardoSpectrum software and setting up your working environment, refer to the *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.

Design Flow

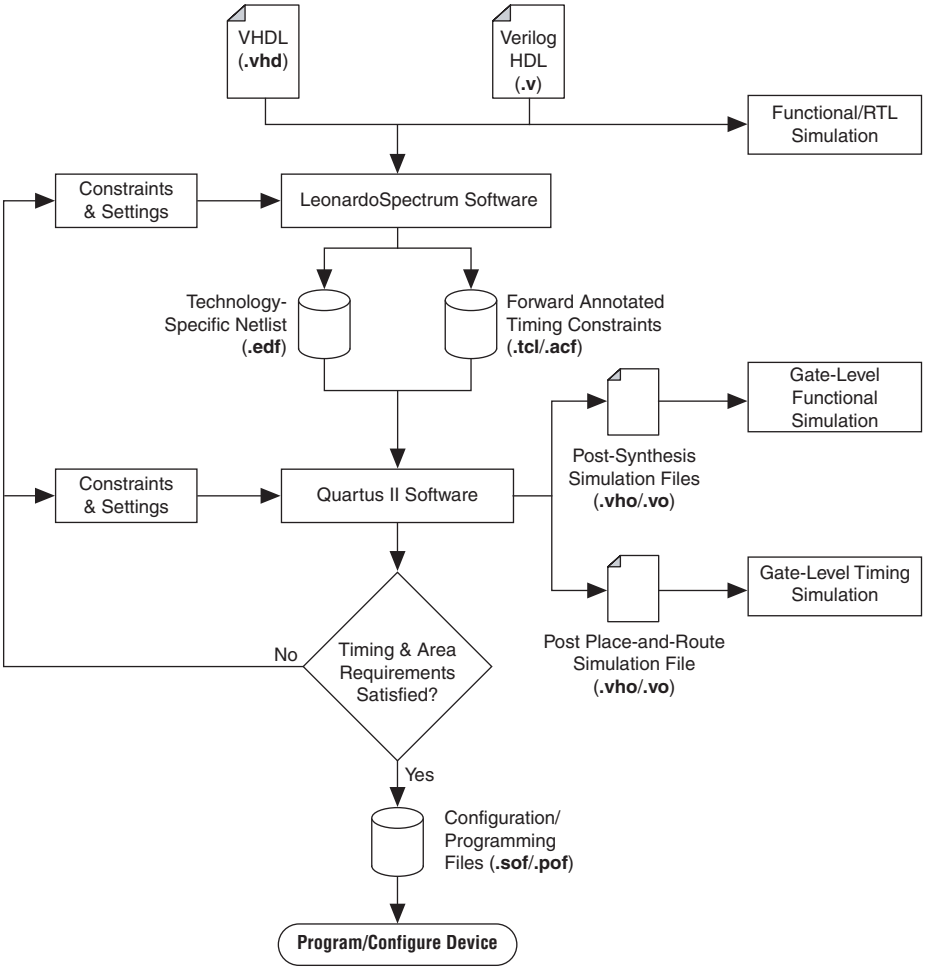
Following are the basic steps in a LeonardoSpectrum-Quartus II design flow:

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 needs, configure or program the Altera device.

Figure 10–1 shows the recommended design flow using the LeonardoSpectrum and Quartus II software.

If your area and timing requirements are satisfied, use the programming files generated from the Quartus II software to program or configure the Altera device. As shown in Figure 10–1, if the area or timing requirements are not met, change the constraints in the LeonardoSpectrum software and re-run the synthesis. Repeat the process until the area and timing requirements are met. You can also use other Quartus II software options and techniques to meet the area and timing requirements.

Figure 10–1. Recommended Design Flow Using LeonardoSpectrum & 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 10–1 lists these file extensions with a short description of each file.

Table 10–1. LeonardoSpectrum Intermediate & 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).
.acf/**.tcl** *(1)*	Forward-annotated constraint file containing constraints and assignments.

Note to Table 10–1:

(1) An assignment and configuration (**.acf**) file is created only for ACEX 1K, FLEX series, and MAX series devices. The assignment and configuration file is generated for backward compatibility with the MAX+PLUS® II software. A Tcl Script File (**.tcl**) is generated for the Quartus II software which also contains Tcl commands to create a Quartus II project.

Altera recommends that you do not use project directory names that include spaces. Some file operations in the LeonardoSpectrum software do not work correctly if the path name contains spaces.

Specify timing constraints and compiler directives for the design in the LeonardoSpectrum software, or in a constraint file (**.ctr**). Many of these constraints are forward-annotated in the Tcl file for use by the Quartus II software.

The LeonardoInsight™ Schematic Viewer is an add-on graphical tool for schematic views of the technology-independent RTL netlist (**.xdb**) and the technology-specific gate-level results. You can use the Schematic Viewer to visually analyze and debug the design. It also supports cross probing between the RTL and gate-level schematics, the design browser, and the source code in the HDLInventor™ text editor.

Optimization Strategies

You can configure most general settings in the **Quick Setup** tab in the LeonardoSpectrum user interface. Other Flow tabs provide additional options, and some Flow tabs include multiple Power tabs (at the bottom of the screen) with still more options. Advanced optimization options in the LeonardoSpectrum software include timing-driven synthesis, encoding style, resource sharing, and mapping I/O registers.

Timing-Driven Synthesis

The LeonardoSpectrum software supports timing-driven synthesis through user-assigned timing constraints to optimize the performance of the design. Setting constraints in the LeonardoSpectrum software are 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 the "Global Power Tab" section are set in the **Constraints** Flow tab. In this tab, there are Power tabs at the bottom, such as **Global** and **Clock**, for setting various constraints.

Global Power Tab

The **Global** tab is the default Power tab in the **Constraints** Flow tab. Specify the global clock frequency here. The **Clock Frequency** on the **Quick Setup** tab is equivalent to the **Registers to Registers** delay setting. You can also specify the following: **Input Ports to Registers**, **Registers to Output Ports**, and **Inputs to Outputs** delays that correspond to global t_{SU}, t_{CO}, and t_{PD} requirements, respectively, in the Quartus II software. The timing diagram on this tab reflects the settings you have made.

Clock Power Tab

You can set various constraints for each clock in your design. First, select the clock name in the **Clock(s)** window. The clock names appear after the design is read from the **Input** Flow tab. Configure settings for that particular clock and click **Apply**. If necessary, you can also set the **Duty Cycle** to a value other than the default 50%. The timing diagram shows these settings.

If a clock has an **Offset** from the main clock, which is considered to be time "`0`", this constraint corresponds to the `OFFSET_FROM_BASE_CLOCK` setting in the Quartus II software.

You can specify the pin number for the clock input pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Input & Output Power Tabs

Configure settings for individual input or output pins in the **Input** and **Output** tabs. First, select a name in the **Input Ports** or **Output Ports** window. The names appear after the design is read from the **Input** Flow tab. Then make the setting for that pin as described below.

The **Arrival Time** setting indicates that the input signal arrives a specified time after the rising clock edge (time "0"). This setting constrains the path from the pin to the first register by including the arrival time in the total delay, and corresponds to the `EXTERNAL_INPUT_DELAY` assignment in the Quartus II software.

The **Required Time** setting indicates the maximum delay after time "0" that the output signal should arrive at the output pin. This setting directly constrains the register to output delay, and corresponds with the `EXTERNAL_OUTPUT_DELAY` assignment in the Quartus II software.

Specify the pin number for the I/O pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Other Constraints

The following sections describe other constraints that can be set with the LeonardoSpectrum user interface.

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. Once 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 10–2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 10–2. State Machine Encoding Styles in the LeonardoSpectrum Software

Style	Description
Binary	Generates state machines with the fewest possible flipflops. Binary state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitchless.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Random	Generates state machines using random state machine encoding. Only use random state machine encoding when no other implementation achieves the desired results.
Auto (default)	Implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

The **Encoding Style** setting is created in the **Input** Flow tab. It instructs the software to use a particular state machine encoding style for all state machines. The default **Auto** selection implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

To ensure proper recognition and improve performance when coding state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* for design guidelines.

Resource Sharing

You can also enable the **Resource Sharing** setting in the **Input** Flow tab. This setting allows optimization to reduce device resources. You should generally leave this setting turned on.

Mapping I/O Registers

The **Map I/O Registers** option is located in the **Technology** Flow tab. The **Map I/O Registers** option applies to Altera FPGAs containing I/O cells (IOCs) or I/O elements (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 Leonardo-Spectrum Software

The LeonardoSpectrum software reports successful synthesis with an information message in the **Transcript** or **Information** window. Estimated device usage and timing results are reported in the Device Utilization section of this window. Figure 10–2 shows an example of a LeonardoSpectrum compilation report.

Figure 10–2. 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" on page 10–10 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. NativeLink integration allows you to run the Quartus II software from within the LeonardoSpectrum software GUI, or to run the LeonardoSpectrum software from within the Quartus II software GUI for device families supported in the Quartus II software.

Generating Netlist Files

The LeonardoSpectrum software generates an EDIF netlist file readable as an input file in the Quartus II software for place-and-route. Select the EDIF file option name in the **Output** Flow tab. The EDIF netlist is also generated if the **Auto** option is turned on in the **Output** Flow tab.

Including Design Files for Black-Boxed Modules

If the design has black-boxed megafunctions, be sure to include the MegaWizard® Plug-In Manager-generated custom megafunction variation design file in the Quartus II project directory, or add it to the list of project files for place-and-route.

Passing Constraints with Scripts

The LeonardoSpectrum software can write out a Tcl file called <***project name***>**.tcl**. This file contains commands to create a Quartus II project along with constraints and other assignments. To output a Tcl script, turn on the **Write Vendor Constraint Files** option in the **Output** Flow tab.

To create and compile a Quartus II project using the Tcl file generated from the LeonardoSpectrum software, perform the following steps in the Quartus II software:

1. Place the EDIF netlist files and Tcl scripts in the same directory.
2. On the View menu, point to Utility, and click **Tcl Console** to open the Quartus II Tcl Console.
3. Type `source` <*path*>/<*project name*>`.tcl` ↵, at a **Tcl Console** command prompt.
4. On the File menu, click **Open Project** to open the new project. On the Processing menu, click **Start Compilation**.

Integration with the Quartus II Software

The **Place And Route** section in the **Quick Setup** tab allows you to launch the Quartus II software from within the LeonardoSpectrum software. Turn on the **Run Integrated Place and Route** option to start the compilation using the Quartus II software to show the fitting and performance results. You can also run the place-and-route software by turning on the **Run Quartus** option on the **Physical** Flow tab and clicking **Run PR**.

To use integrated place-and-route software, on the Options menu, point to Place and Route Path and click **Tools**, and specify the location of the Quartus II software executable file (browse to <*Quartus II software installation directory*>**/bin)**.

Guidelines for Altera Megafunctions & 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^{sm}).

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 setup 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 the black box methodology.

Altera recommends using the megafunction wizard to ensure that the ports and parameters are set correctly.

When directly instantiating megafunctions, see the Quartus II Help for a list of the ports and parameters.

Inferring Altera Memory Elements

The LeonardoSpectrum software can infer memory blocks from Verilog HDL or VHDL code. When the LeonardoSpectrum software detects a RAM or ROM from the style of the RTL code at a technology-independent level, it then maps the element to a generic module in the RTL database. During the technology-mapping phase of synthesis, the LeonardoSpectrum software maps the generic module to the most optimal primitive memory cells, or Altera megafunction, for the target Altera technology.

For more information about inferring RAM and ROM megafunctions, including examples of VHDL and Verilog HDL code, see the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring RAM

The LeonardoSpectrum software supports RAM inference for various device families. The restrictions for the LeonardoSpectrum software to successfully infer RAM in a design are listed below:

- The write process must be synchronous
- The read process can be asynchronous or synchronous depending on the target Altera architecture
- Resets on the memory are not supported

Table 10–3 shows a summary of the minimum memory sizes and minimum address widths for inferring RAM in various device families.

To disable RAM inference, set the `extract_ram` and `infer_ram` variables to "false." On the Tools menu, click **Variable Editor** to enter the value "false" when synthesizing in the user interface with the Advanced Flow tabs, or add the commands `set extract_ram false` and `set infer_ram false` to your synthesis script.

Table 10–3. Inferring RAM Summary

	Stratix II, Stratix, Stratix GX & Cyclone Series	APEX Series, Excalibur & Mercury	FLEX 10KE & ACEX 1K
RAM primitive	`altsyncram`	`altdpram`	`altdpram`
Minimum RAM size	2 bits	64 bits	128 bits
Minimum address width	1 bit	4 bits	5 bits

Inferring ROM

You can implement ROM behavior in HDL source code with `CASE` statements or specify the ROM as a table. The LeonardoSpectrum software infers both synchronous and asynchronous ROM depending on the target Altera device. For example, memory for the Stratix series devices must be synchronous to be inferred.

To disable ROM inference, set the `extract_rom` variable to "false." To enter the value "false" when synthesizing in the user interface with the **Advanced Flow** tabs, on the Tools menu, click **Variable Editor**, or add the commands `set extract_rom false` to your synthesis script.

Inferring Multipliers & 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-boxing 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

In devices that include dedicated DSP blocks, multipliers, multiply-accumulators, and multiply-adders can be implemented either in DSP blocks or in logic. You can control this implementation through attribute settings in the LeonardoSpectrum software.

As shown in Table 10–4, attribute settings in the LeonardoSpectrum software control the implementation of the multipliers in DSP blocks or logic at the signal block (or module), and project level.

Table 10–4. Attribute Settings for DSP Blocks in the LeonardoSpectrum Software *Note (1)*

Level	Attribute Name	Value	Description
Global	extract_mac *(2)*	TRUE	All multipliers in the project mapped to DSP blocks.
		FALSE	All multipliers in the project mapped to logic.
Module	extract_mac *(3)*	TRUE	Multipliers inside the specified module mapped to DSP blocks.
		FALSE	Multipliers inside the specified module mapped to logic.
Signal	dedicated_mult	ON	LPM inferred and multipliers implemented in DSP block.
		OFF	LPM inferred, but multipliers implemented in logic by the Quartus II software.
		LCELL	LPM not inferred, and multipliers implemented in logic by the LeonardoSpectrum software.
		AUTO	LPM inferred, but the Quartus II software automatically maps the multipliers to either logic or DSP blocks based on the Quartus II software place-and-route.

Notes to Table 10–4:

(1) The extract_mac attribute takes precedence over the dedicated_mult attribute.
(2) For devices with DSP blocks, the extract_mac attribute is set to "true" by default for the entire project.
(3) For devices with DSP blocks, the extract_mac attribute is set to "true" by default for all modules.

Global Attribute

You can set the global attribute extract_mac to control the implementation of multipliers in DSP blocks for the entire project. You can set this attribute using the script interface. The script command is:

```
set extract_mac <value>
```

Module Level Attributes

You can control the implementation of multipliers inside a module or component by setting attributes in the Verilog HDL source code. The attribute used is extract_mac. Setting this attribute for a module affects only the multipliers inside that module. The command is:

```
//synthesis attribute <module name> extract_mac <value>
```

The Verilog HDL and VHDL codes samples shown in Examples 10–1 and 10–2 show how to use the `extract_mac` attribute.

Example 10–1. 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 10–2. Using Module Level Attributes in VHDL Code

```
library ieee ;
USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

entity mult_acc is
   generic (size : integer := 4) ;
   port (
      a: in std_logic_vector (size-1 downto 0) ;
      b: in std_logic_vector (size-1 downto 0) ;
      clk : in std_logic;
        accum_out: inout std_logic_vector (2*size downto 0)
   ) ;
   attribute extract_mac : boolean;
   attribute extract_mac of mult_acc : entity is FALSE;
end mult_acc;

architecture synthesis of mult_acc is
    signal a_int, b_int : signed (size-1 downto 0);
    signal pdt_int : signed (2*size-1 downto 0);
    signal adder_out : signed (2*size downto 0);

begin
    a_int <= signed (a);
    b_int <= signed (b);
    pdt_int <= a_int * b_int;
    adder_out <= pdt_int + signed(accum_out);
    process (clk)
    begin
       if (clk'event and clk = '1') then
          accum_out <= std_logic_vector (adder_out);
       end if;
    end process;
end synthesis ;
```

Signal Level Attributes

You can control the implementation of individual `lpm_mult` multipliers by using the `dedicated_mult` attribute as shown below:

```
//synthesis attribute <signal_name> dedicated_mult <value>
```

The `dedicated_mult` attribute is only applicable to signals or wires; it is not applicable to registers.

Table 10–5 shows the supported values for the dedicated_mult attribute.

Table 10–5. 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 10–5:

(1) Although both dedicated_mult=OFF and dedicated_mult=LCELLS result in logic implementations, the optimized results in these two cases may differ.

Some signals for which the dedicated_mult attribute is set may 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.

Examples 10–3 and 10–4 are samples of Verilog HDL and VHDL codes, respectively, using the dedicated_mult attribute.

Example 10–3. 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 10–4. 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-boxing 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 may be lost or data may 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, then 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 may get a no fit error in the Quartus II software. In such cases, use the attribute settings in the LeonardoSpectrum software to control the mapping of multipliers in your design to DSP blocks or logic.

Block-Based Design with the Quartus II Software

The incremental compilation and LogicLock™ block-based design flows enable users to design, optimize, and lock down a design one section at a time. You can independently create and implement each logic module into a hierarchical or team-based design. With this method, you can preserve the performance of each module during system integration and have more control over placement of your design. To maximize the benefits of the incremental compilation or LogicLock design methodology in the Quartus II software, you can partition a new design into a hierarchy of netlist files during synthesis in the LeonardoSpectrum software.

The LeonardoSpectrum software allows you to create different netlist files for different sections of a design hierarchy. Different netlist files mean 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 & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about the LogicLock feature, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook.*

Hierarchy & 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* 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. If two different modules are in the same design file but are defined as being part of different regions, it is difficult to maintain incremental synthesis since both regions would 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 requires optimizing through hierarchies, lower-level tri-states are not supported with a block-level design methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

If the hierarchy is flattened during synthesis, logic is optimized across boundaries, preventing you from making LogicLock assignments to the flattened blocks. Altera recommends preserving the hierarchy when compiling the design. In the **Optimize** command of your script, use the **Hierarchy Preserve** command or in the user interface select **Preserve** in the **Hierarchy** section on the **Optimize** Flow tab.

If you are compiling your design with a script, you can use an alternative method for preventing optimization across boundaries. In this case, use the **Auto** hierarchy setting and set the `auto_dissolve` attribute to false on the instances or views that you want to preserve (that is, the modules with LogicLock assignments) using the following syntax:

```
set_attribute -name auto_dissolve -value false
    .work.<block1>.INTERFACE
```

This alternative method flattens your design according to the `auto_dissolve` limits, but does not optimize across boundaries where you apply the attribute as described.

For more details on LeonardoSpectrum attributes and hierarchy levels, refer to the LeonardoSpectrum documentation in the Help menu.

Creating a Design with Multiple EDIF Files

The first stage of a hierarchical design flow is to generate multiple EDIF files, enabling you to take advantage of the incremental compilation flows in the Quartus II software. If the whole design is in one EDIF file, changes in one block affect other blocks because of possible node name changes. You can generate multiple EDIF files either by using the LogicLock option in the LeonardoSpectrum software, or by manually black boxing each block that you want to be part of a LogicLock region.

Once you have created multiple EDIF files using one of these methods, you must create the appropriate Quartus II project(s) to place-and-route the design.

Generating Multiple EDIF Files Using the LogicLock Option

This section describes how to generate multiple EDIF files using the LogicLock option in the LeonardoSpectrum software. When synthesizing a top-level design that includes LogicLock regions, use 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, use the following steps in the LeonardoSpectrum software:

1. Switch to the **Advanced** Flow tab instead of the **Quick Setup** tab (Tools menu).
2. Set the target technology and speed grade for the device on the **Technology** Flow tab.
3. Open the input source files on the **Input** Flow tab.
4. Click **Read** on the **Input** Flow tab to read the source files but not begin optimization.
5. Select the **Module** Power tab located at the bottom of the **Constraints** Flow tab.
6. Click on a module to be placed in a LogicLock region in the **Modules** section.
7. Turn on the **LogicLock** option.
8. Type the desired LogicLock region name in the text field under the **LogicLock** option.
9. Click **Apply**.
10. Repeat steps 6-9 for any other modules that you want to place in LogicLock regions.

In some cases, you are prompted to save your LogicLock and other non-global constraints in a Constraints File (**.ctr**) when you click anywhere off the **Constraints** Flow tab. The default name is <*project name*>**.ctr**. This file is added to your **Input** file list, and must be manually included later if you recreate the project.

The command written into the LeonardoSpectrum Information or Transcript Window is the Tcl command that gets written into the CTR file. The format of the "path" for the module specified in the command should be work.<*module*>.**INTERFACE**. To ensure that you don't see an optimized version of the module, do not perform a **Run Flow** on the **Quick Setup** tab prior to setting LogicLock constraints. Always use the **Read** command, as described in step 4.

11. Continue making any other settings as required on the **Constraints** tab.

12. Select **Preserve** in the **Hierarchy** section on the **Optimize** tab to ensure that the hierarchy names are not flattened during optimization.

13. Continue making any other settings as required on the **Optimize** tab.

14. Run your synthesis flow using each Flow tab, or click **Run Flow**.

Synthesis creates an EDIF file for each module that has a LogicLock assignment in the **Constraints** Flow tab. You can now use these files with the incremental compilation flows in the Quartus II software.

You might occasionally see multiple EDIF files and LogicLock commands for the same module. An "unfolded" version of a module is created when you instantiate a module more than once and the boundary conditions of the instances are different. For example, if you apply a constant to one instance of the block, it might be optimized to eliminate unneeded logic. In this case, the LeonardoSpectrum software must create a separate module for each instantiation (unfolding). If this unfolding occurs, you see more than one EDIF file, and each EDIF file has a LogicLock assignment to the same LogicLock region. When you import the EDIF files to the Quartus II software, the EDIF files created from the module are placed in different LogicLock regions. Any optimizations performed in the Quartus II software using the LogicLock methodology must be performed separately for each EDIF netlist.

Creating a Quartus II Project for Multiple EDIF Files Including LogicLock Regions

The LeonardoSpectrum software creates Tcl files that provide the Quartus II software with the appropriate LogicLock assignments, creating a region for each EDIF file along with the information to set up a Quartus II project.

The Tcl file contains the commands shown in Example 10–5 for each LogicLock region. This example is for module `taps` where the name `taps_region` was typed as the LogicLock region name in the **Constraints** Flow tab in the LeonardoSpectrum software.

Example 10–5. Tcl File for Module Taps with taps_region as LogicLock Region Name

```
project add_assignment {taps} {taps_region} {} {}
    {LL_AUTO_SIZE} {ON}
project add_assignment {taps} {taps_region} {} {}
    {LL_STATE} {FLOATING}
project add_assignment {taps} {taps_region} {} {}
    {LL_MEMBER_OF} {taps_region}
```

These commands create a LogicLock region with Auto-Size and Floating-Origin properties. This flexible LogicLock region allows the Quartus II Compiler to select the size and location of the region.

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

You can use the following methods to import the EDIF file and corresponding Tcl file into the Quartus II software:

- Use the Tcl file that is created for each EDIF file by the LeonardoSpectrum software. This method allows you to generate multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Altera recommends this method for bottom-up incremental and hierarchical design methodologies because it allows each block in the design to be treated separately. Each block can be brought into one top-level project with the import function.

  or

- Use the <*top-level project*>**.tcl** file that contains the assignments for all blocks in the project. This method allows the top-level designer to import all the blocks into one Quartus II project. You can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can use their EDIF file to create a separate project at that time. You would then have to add new assignments to the top-level project using the import function.

In both methods, you can use the following steps to create the Quartus II project, import the appropriate LogicLock assignments, and compile the design:

1. Place the EDIF and Tcl files in the same directory.
2. On the View menu, point to Utility Windows and click **Tcl Console** to open the Quartus II **Tcl Console**.

3. Type `source <path>/<project name>.tcl ↵`.

4. To open the new completed project, on the File menu, click **Open Project**. Browse to and select the project name, and click **Open**.

For more information about importing design using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock assignments, see the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

Generating Multiple EDIF Files Using Black Boxes

This section describes how to manually generate multiple EDIF files using the black-boxing technique. The manual flow, described below, was supported in older versions of the LeonardoSpectrum software. The manual flow is discussed here because some designers want more control over the project for each submodule.

To create multiple EDIF files in the LeonardoSpectrum software, create a separate project for each module and top-level design that you want to maintain as a separate EDIF file. Implement black-box instantiations of lower-level modules in your top-level project.

When synthesizing the projects for the lower-level modules and the top-level design, use the following general guidelines.

For lower-level modules:

- Turn off **Map IO Registers** for the target technology on the **Technology** Flow tab.
- Read the HDL files for the modules. Modules may include black-box instantiations of lower-level modules that are also maintained as separate EDIF files.
- Add constraints.
- Turn off **Add I/O Pads** on the **Optimize** Flow tab.

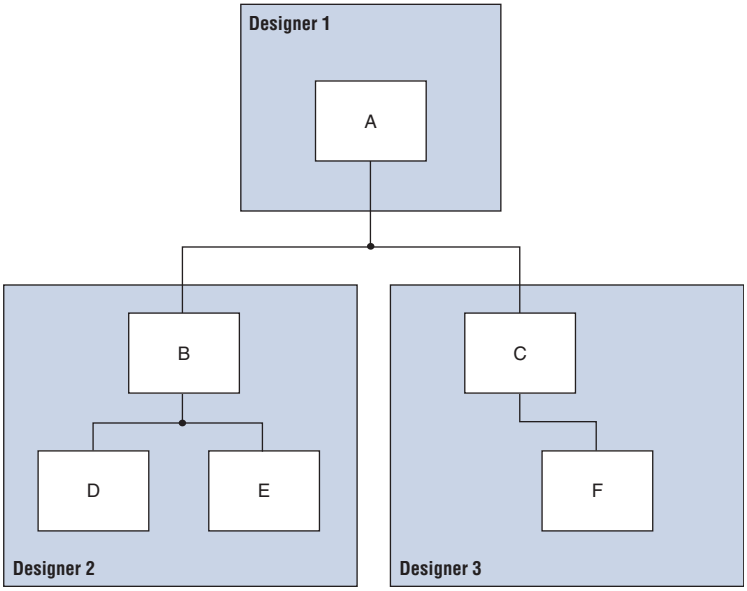
For the top-level design:

- Turn on **Map IO Registers** if you want to implement input and/or output registers in the IOEs for the target technology on the **Technology** Flow tab.
- Read the HDL files for the top-level design.
  - Black-box lower-level modules in the top-level design
- Add constraints (clock settings should be made at this time).

The following sections describe examples of black-box modules in a block-based and team-based design flow.

In Figure 10–3, 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 10–3. Block-Based & 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, while another netlist is created for C and its submodule F. To create multiple EDIF files, perform the following steps:

1. Generate an EDIF file for module C. Use **C.v** and **F.v** as the source files.
2. Generate an EDIF file for module B. Use **B.v**, **D.v**, and **E.v** as the source files.
3. Generate a top-level EDIF file **A.v** for module A. Ensure that your black-box modules B and C were optimized separately in steps 1 and 2.

Black Boxing 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 10–6 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 10–6. 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 instructs 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, you need a component declaration for the black box which is normal for any other block in the design.

Example 10–7 shows an example of the **A.vhd** top-level file. If any of your lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 10–7. VHDL Top-Level File Black-Boxing Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15
);
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT  INTEGER RANGE 0 TO 15
);
END COMPONENT;

COMPONENT C PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

-- Other component declarations in A.vhd go here

signal cnt_out : INTEGER RANGE 0 TO 15;
signal reg_a_out : INTEGER RANGE 0 TO 15;
BEGIN
CNT : C
PORT MAP (
    data_in => data_in,
    clk => clk,
    e => e,
    ld => ld,
    data_out => cnt_out
);

REG_A : D
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => reg_a_out
);

-- Any other code in A.vhd goes here

END a_arch;
```

Previous versions of the LeonardoSpectrum software required the attribute statement `noopt of C: component is TRUE`, which instructed the software to treat the component C as a black box. This attribute is no longer required, although it is still supported in the software.

After you have completed the steps outlined in this section, you have a different EDIF netlist file for each block of code. You can now use these files for incremental compilation flows in the Quartus II software.

Creating a Quartus II Project for Multiple EDIF Files

The LeonardoSpectrum software creates a Tcl file for each EDIF file, which provides the Quartus II software with the information to set up a project.

As in the previous section, there are two different methods for bringing each EDIF and corresponding Tcl file into the Quartus II software:

- Use the Tcl file that is created for each EDIF file by the LeonardoSpectrum software. This method generates multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Altera recommends this method for bottom-up incremental and hierarchical design methodology because it allows each block in the design to be treated separately; each block can be imported into one top-level project.

  or

- Use the <*top-level project*>**.tcl** file that contains the information to set up the top-level project. This method allows the top-level designer to create LogicLock regions for each block and bring all the blocks into one Quartus II project. Designers can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can take their EDIF file and create a separate Quartus II project at that time. New assignments would then have to be added to the top-level project manually or through the import function.

For more information about importing designs using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock regions, refer to the *LogicLock Design Methodology* chapter in the volume 2 of the *Quartus II Handbook*.

In both methods, you can use the following steps to create the Quartus II project and compile the design:

1. Place the EDIF and Tcl files in the same directory.
2. On the View menu, point to Utility Windows and click **Tcl Console**. The Quartus II **Tcl Console** is shown.
3. At a Tcl prompt, type `source` <*path*>/<*project name*>`.tcl` ↵.
4. On the File menu, click **Open Project**. In the **New Project** window, browse to and select the project name. Click **Open**.
5. To create LogicLock assignments, on the Assignments menu, click **LogicLock Regions Window**.
6. On the Processing menu, click **Start Compilation**.

Incremental Synthesis Flow

If you make changes to one or more submodules, you can manually create new projects in the LeonardoSpectrum software to generate a new EDIF netlist file when there are changes to the source files. Alternatively, you can use incremental synthesis to generate a new netlist for the changed submodule(s). To perform incremental synthesis in the LeonardoSpectrum software, use the script described in this section to reoptimize and generate a new EDIF netlist for only the affected modules using the LeonardoSpectrum top-level project. This method applies only when you are using the **LogicLock** option in the LeonardoSpectrum software.

Modifications Required for the LogicLock_Incremental.tcl Script File

There are three sets of entries in the file that must be modified before beginning incremental synthesis. The variables in the Tcl file are surrounded by angle brackets (< >).

1. Add the list of source files that are included in the project. You can enter the full path to the file or just the file name if the files are located in the working directory.

2. Indicate which modules in the design have changed. These modules are the EDIF files that are regenerated by the LeonardoSpectrum software. These modules contain a LogicLock assignment in the original compilation.

   Obtain the LeonardoSpectrum software path for each module by looking at the CTR file that contains the LogicLock assignments from the original project. Each LogicLock assignment is applied to a particular module in the design.

3. Enter the target device family using the appropriate device keyword. The device keyword is written into the **Transcript** or **Information** window when you select a target Technology and click **Load Library** or **Apply** on the **Technology Flow** tab in the graphical user interface.

Example 10–8 shows the **LogicLock_Incremental.tcl** file for the incremental synthesis flow. You must modify the Tcl file before you can use it for your project.

Example 10–8. 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

Once you have modified the Tcl script, as described in the "Modifications Required for the LogicLock_Incremental.tcl Script File" on page 10–31, you can compile your design using the script.

You can run the script in batch mode at the command line prompt using the following command:

```
spectrum -file <Tcl_file> ↵
```

To run the script from the interface, on the File menu, click **Run Script**, then browse to your Tcl file and click **Open**.

The LogicLock incremental design flow uses module-based design to help you preserve performance of modules and have control over placement. By tagging the modules that require separate EDIF files, you can make multiple EDIF files for use with the Quartus II software from a single LeonardoSpectrum software project.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Mentor Graphics LeonardoSpectrum software and the Quartus II design flow allows you to control how your design files are prepared for the Quartus II place-and-route process, as well as to improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Document Revision History

Table 10–6 shows the revision history of this document.

Table 10–6. Document Revision History

Date & Documentation Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Added document revision history to chapter.	
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 10 was formerly chapter 11 in version 5.0.	
May 2005 v5.0.0	Chapter 11 was formerly chapter 9 in version 4.2.	
Dec. 2004 v2.1	• Chapter 10 was formerly Chapter 11 in version 4.1. • Updated information. • New functionality in Quartus II software version 4.2. • Updated tables and figures.	
June 2004 v2.0	• Updates to tables, and figures. • New functionality for Quartus II software version 4.1.	
Feb. 2004 v1.0	Initial release.	

11. Mentor Graphics Precision RTL Synthesis Support

QII51011-6.1.0

Introduction

As programmable logic device (PLD) designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. This chapter documents support for the Mentor Graphics® Precision RTL Synthesis software in the Quartus® II software design flow, as well as key design methodologies and techniques for improving your results for Altera® devices. This chapter includes the following sections:

- General design flow with the Precision RTL Synthesis software and the Quartus II software
- Creating a project and compiling the design
- Setting constraints to achieve optimal results
- Synthesizing the design and evaluating the results
- Exporting designs to the Quartus II software using NativeLink® integration
- Guidelines for Altera megafunctions and the library of parameterized modules (LPM) functions, instantiating them in a clear-box or black-box flow using the MegaWizard® Plug-In manager, and tips for inferring them from HDL code
- Incremental compilation and block-based design

This chapter assumes that you have installed and licensed the Precision RTL Synthesis software and the Quartus II software.

To obtain and license the Precision RTL Synthesis software, refer to the Mentor Graphics web site at **www.mentor.com**. To install and run the Precision RTL Synthesis software and to set up your work environment, refer to the *Precision RTL Synthesis User's Manual* in the Precision Manuals Bookcase in the Help menu.

Design Flow

The basic steps in a Quartus II design flow using the Precision RTL Synthesis software are as follows:

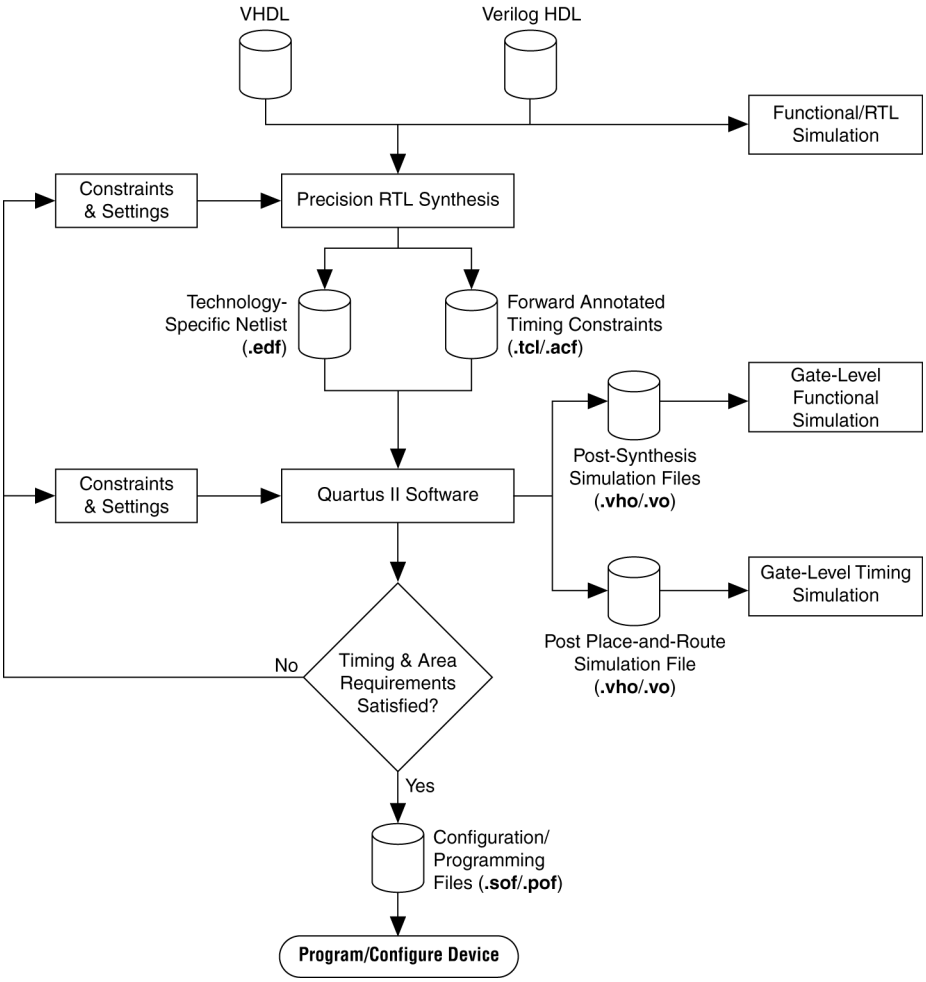
1. Create Verilog HDL or VHDL design files in the Quartus II design software, the Precision RTL Synthesis software, or with a text editor.
2. Create a project in the Precision RTL Synthesis software that contains the HDL files for your design, select your target device, and set global constraints. For best results when using Altera megafunctions, Mentor Graphics recommends using the clear box option which enables synthesis to report more accurate resource utilization and timing estimates. Refer to "Clear-Box Methodology" on page 11–20 for details.
3. Compile the project in the Precision RTL 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 RTL Synthesis software. With the design analysis capabilities and cross-probing of Precision RTL 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 technology-specific EDIF (**.edf**) netlist and the tool command language (**.tcl**) file generated by the Precision RTL Synthesis software into the Quartus II software for placement and routing, and for performance evaluation using actual post-layout timing data.
7. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

These steps are described in detail throughout this chapter. Figure 11–1 shows the Quartus II design flow using Precision RTL Synthesis as described in the steps above.

Figure 11–1. Design Flow Using the Precision RTL Synthesis Software & Quartus II Software



If your area or timing requirements are not met, you can change the constraints and resynthesize the design in the Precision RTL 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 (Figure 11–1).

You can use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is the **WYSIWYG Primitive Resynthesis** option, which can perform optimizations on your EDIF netlist in the Quartus II software.

For information about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*. For more recommendations on how to optimize your design, refer to the *Area & 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 RTL Synthesis software produces several intermediate and output files. Table 11–1 lists these files with a short description of each file type.

Table 11–1. Precision RTL Synthesis Software Intermediate & Output Files

File Extension(s)	File Description
.sdc	Design constraints file in Synopsys Design Constraints File
.psp	Precision RTL Synthesis Software Project File
.xdb	Mentor Graphics Design Database File
.rep *(1)*	Synthesis Area & Timing Report File
.edf	Technology-specific netlist in electronic design interchange format (EDIF)
.acf/**.tcl** *(2)*	Forward-annotated constraints file containing constraints and assignments

Notes to Table 11–1:

(1) The timing report file includes performance estimates that are based on preplace-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 may 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 & Evaluating the Results" on page 11–11* for details.

(2) An Assignment & Configuration File (**.acf**) file is created only for ACEX® 1K, FLEX® 10K, FLEX 10KA, FLEX 6000, FLEX 8000, MAX® 7000, MAX 9000, and MAX 3000 devices. The Assignment & Configuration File is generated for backward compatibility with the MAX+PLUS® II software. A Tcl file for the Quartus II software is created for all devices, which also contains Tcl commands to create and compile a Quartus II project.

Creating a Project & Compiling the Design

After creating your design files, create a project in the Precision RTL 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 RTL Synthesis software, click the **New Project** icon in the Design Bar on the left side of the GUI.
2. Set the **Project Name** and the **Project Folder**. The implementation name of the design corresponds to this project name.
3. Add input files to the project with the **Add Input Files** icon in the Design Bar. Precision RTL 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. All timing constraints are forward-annotated to the Quartus II software using Tcl scripts.

To generate additional netlist files (for example, an HDL netlist for simulation), on the Tools menu, point to **Set Options > Output** and click **Additional Output Netlist**. The Precision RTL Synthesis software generates a separate file for each selected type of file: EDIF, Verilog HDL, and VHDL.

Compiling the Design

To compile the design into a technology-independent implementation, click the **Compile** icon in the Design Bar.

Setting Constraints

In the next steps, you set constraints and map the design to technology-specific cells. The Precision RTL 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 RTL Synthesis software performs static timing analysis to determine the location of the critical timing paths. The Precision RTL Synthesis software achieves the best results for your design when you set as many realistic constraints as possible. Ensure to set constraints for timing, mapping, false paths, multicycle paths, and others that control the structure of the implemented design.

Mentor Graphics recommends creating a Synopsys Design Constraint file (**.sdc**) and adding this file to the Constraint Files section of the Project Files list. You can create this file with a text editor or use the Precision RTL Synthesis software to generate one automatically for you on the first synthesis run. To create an initial constraint file manually, set constraints on design objects (such as clocks, design blocks, or pins) in the Design Hierarchy browser. By default, the Precision RTL 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 **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.

For more details and examples, refer to the Attributes chapter in the *Precision Synthesis Reference Manual* in the Precision Manual Bookcase in the Help menu.

Setting Timing Constraints

Timing constraints, based on the industry-standard Synopsys Design Constraint file format, help the Precision RTL Synthesis software to deliver optimal results. Missing timing constraints can result in incomplete timing analysis and may prevent timing errors from being detected. Precision RTL Synthesis software provides constraint analysis prior to synthesis to ensure that designs are fully and accurately constrained. All timing constraints are forward-annotated to the Quartus II software using Tcl scripts.

Because the Synopsys Design Constraint file format requires that timing constraints must be set relative to defined clocks, you must specify your clock constraints before applying any other timing constraints.

You also can use multicycle path and false path assignments to relax requirements or exclude nodes from timing requirements. Doing so can improve area utilization and allow the software optimizations to focus on the most critical parts of the design.

For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis Users Manual* and the *Precision Synthesis Reference Manual* available in the Precision Manual Bookcase in the Help menu.

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 & I/O Settings

The Precision RTL 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 written into the 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 the Synopsys Design Constraint file 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 Synopsys Design Constraint file.

Table 11–2. Constraint File Settings

Constraint	Entry Format for Synopsys Design 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 RTL 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 **SLOWSLEW** lists.

You also can use synthesis attributes or pragmas in your HDL code to make these assignments. The following code samples show you how to 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 them in your HDL code, refer to the *Precision Synthesis Reference Manual*.

Assigning I/O Registers

The Precision RTL Synthesis software performs timing-driven I/O register mapping by default. It moves registers into an I/O element (IOE) when it does not negatively impact the register-to-register performance of your design, based on the timing constraints.

You can force a register to the device's IOE using the Complex I/O constraint. This option does not apply if you turn off I/O pad insertion. Refer to "Disabling I/O Pad Insertion" for more information.

To force an I/O register into the device's IOE using the GUI, follow these steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy browser.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry, as desired.
3. Under **Inputs** or **Outputs**, right-click the desired pin name and select **Force Register into IO**.

You also can make the assignment by right-clicking on the pin in the Schematic Viewer.

For Stratix® II, Cyclone™ II, MAX® II, Stratix, and Cyclone families of devices, the Precision RTL Synthesis software can move an internal register to an I/O register without any restrictions on design hierarchy.

For more mature devices, the Precision RTL 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 RTL Synthesis software assigns I/O pad atoms (device primitives used to represent the I/O pins and I/O registers used) to all ports in the top level of a design by default. In certain situations, you may 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 RTL Synthesis software with the most information about the top-level pins in the design.

Preventing the Precision RTL Synthesis Software from Adding I/O Pads

If you are compiling a subdesign as a separate project, I/O pins cannot be primary inputs or outputs of the device and therefore should not have an I/O pad associated with them. To prevent the Precision RTL Synthesis software from adding I/O pads, perform the following steps:

1. On the Tools menu, click **Set Options**.
2. On the **Optimization** page of the **Options** dialog box, turn off **Add IO Pads**, then click **Apply**.

This procedure adds the following command to the project file:

```
setup_design -addio=false
```

Preventing the Precision RTL 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 Double Data Rate (DDR) or a Phase-Locked Loop (PLL), at the external ports of the design, follow these steps:

1. After compiling the design, in the Design Hierarchy browser, expand the **Ports** entry by clicking the +.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry.
3. Under **Inputs** or **Outputs**, right-click the desired pin name and click **Set Input Constraints**.
4. In the **Port Constraints** dialog box for the selected pin name, turn off **Insert Pad**.

You also can make the assignment by right-clicking on the pin in the Schematic Viewer or by attaching the `nopad` attribute to the port in the HDL source code.

Controlling Fan-Out on Data Nets

Fan-out is defined as the number of nodes driven by an instance or top-level port. High fan-out nets can have significant delays which can result in an unroutable net. On a critical path, high fan-out nets can cause larger delay in a single net segment which can 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 RTL 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 RTL 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 & Evaluating the Results

To synthesize the design for the target device, click on the **Synthesize** icon in the Precision RTL Synthesis Design Bar. During synthesis, the Precision RTL Synthesis software optimizes the compiled design, then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the naming convention:

<project name>`_impl_`*<number>*

After synthesis is complete, you can evaluate the results in terms of area and timing. The *Precision RTL Synthesis User's Manual* on the Mentor Graphics web site describes different results that can be evaluated in the software.

There are several schematic viewers available in the Precision RTL 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, if needed, to optimize the design.

Obtaining Accurate Logic Utilization & 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 need, and how fast the design will run. However, today's FPGA devices provide a wide variety of advanced features in addition to basic registers and look-up tables. The Quartus II software has advanced algorithms to take advantage of these features, as well as optimization techniques to both increase performance and reduce the amount of logic required for a given design. In addition, designs may contain black boxes and functions that take advantage of specific device features. Because of these advances, synthesis tool reports provide post-synthesis area and timing estimates, but the place-and-route software should be used to obtain final logic utilization and timing reports.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, which allows you to run other EDA design entry/synthesis, simulation, and timing analysis tools automatically from within the Quartus II software.

After a design is synthesized in the Precision RTL 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, written as Tcl scripts. 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 RTL Synthesis software, and makes timing assignments. For many devices, the Project Configuration script calls the place-and-route constraints script, <*project name*>**_pnr_constraints.tcl**, to make your timing constraints.

Running the Quartus II Software from within the Precision RTL Software

Precision RTL 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 RTL 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 Quartus II options. On the Tools menu, click **Set Options**. On the **Integrated Place and Route** page, specify the path to the Quartus II executables in the **Path to Quartus II installation** box.

To automate the place-and-route process, click the **Run Quartus** icon in the **Quartus II** window of the Precision RTL 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, no user interface appears).

Two primary Precision RTL Synthesis software commands control the place-and-route process. Place-and-route options are set by the **setup_place_and_route** command. The process is started with the **place_and_route** command.

Precision RTL Synthesis software versions 2004a and later support using individual Quartus II executables, such as analysis and synthesis (**quartus_map**), Fitter (**quartus_fit**), and Timing Analyzer (**quartus_tan**), for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus II Modular** flow option in Precision RTL Synthesis software and is compatible with Quartus II software versions beginning with version 4.0. By default, the Precision RTL Synthesis software generates this Quartus II Project Configuration File (Tcl file) for Stratix II, Stratix, Stratix GX, MAX II, Cyclone II, and Cyclone device families. When using this flow, all timing constraints that you set during synthesis are exported to the Quartus II place-and-route constraints file (<*project name*>**_pnr_constraints.tcl**).

For other device families, Precision RTL Synthesis software uses the **Quartus II** flow option, which enables the Quartus II compilation flow that existed in Precision RTL Synthesis software versions earlier than 2004a. The Quartus II Project Configuration File (Tcl file) written when using the **Quartus II** flow option includes supported timing constraints that you specified during synthesis. This Tcl file is compatible with all versions of the Quartus II software; however, the format and timing constraints do not take full advantage of the features in the Quartus II software introduced with version 4.0.

To force the use of a particular flow when it is not the default for a certain device family, use the following command to set up the integrated place-and-route flow:

```
setup_place_and_route -flow "<Altera Place-and-Route flow>"
```

Depending on the device family, you can use one of the following flow options in the command above:

- `Quartus II Modular`
- `Quartus II`
- `MAX+PLUS II`

For example, for the Stratix II or MAX II device families (which were not supported in Quartus II software versions earlier than 4.0), you can use only the **Quartus II Modular** flow. For the Stratix device family you can use either the **Quartus II Modular** or **Quartus II** flows. The FLEX 8000 device family, which is not supported in the Quartus II software, is supported only by the **MAX+PLUS II** flow.

After the design is compiled in the Quartus II software from within the Precision RTL Synthesis software, you can invoke the Quartus II GUI manually and then open the project using the generated Quartus II project file. You can view reports, run analysis tools, specify options, and run the various processing flows available in the Quartus II software.

Running the Quartus II Software Manually Using the Precision RTL Synthesis-Generated Tcl Script

You can use the Quartus II software separately from the Precision RTL Synthesis software. To run the Tcl script generated by the Precision RTL Synthesis software to set up your project and start a full compilation, perform the following steps:

1. Ensure the EDIF and Tcl files are located in the same directory (they should both be located in the implementation directory by default).
2. In the Quartus II software, on the View menu, point to Utility Windows and click **Tcl Console**.
3. At the Tcl Console command prompt, type the command:

   `source <path>/<project name>.tcl` ↵

4. On the File menu, click **Open Project**. Browse to the project name, and click **Open**.
5. Compile the project in the Quartus II software.

Using Quartus II Software to Launch the Precision RTL Synthesis Software

Using NativeLink integration, you can set up the Quartus II software to run the Precision RTL Synthesis software. This feature allows you to use the Precision RTL Synthesis software to synthesize a design as part of a normal compilation.

For detailed information about using NativeLink integration with the Precision RTL Synthesis software, go to *Specifying EDA Tool Settings* in the Quartus II Help index.

Passing Constraints to the Quartus II Software

The place-and-route constraints script forward-annotates timing constraints that you made in the Precision RTL Synthesis software. This integration allows you to enter these constraints once in the Precision RTL Synthesis software, and then pass them automatically to the Quartus II software.

The following constraints are translated by the Precision RTL Synthesis software:

- `create_clock`
- `set_input_delay`
- `set_output_delay`
- `set_false_path`
- `set_multicycle_path`

create_clock

You can specify a clock in the Precision RTL 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 always in units of 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 (that is, 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, but `-period` *<period_value>* is specified, the default waveform has a rising edge of 0.0 and a falling edge of *<period_value>*/2.

The Precision RTL Synthesis software passes the clock definitions to the Quartus II software with the `create_base_clock` command.

The following list describes some differences in the clock properties supported by the Precision RTL 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 RTL Synthesis software finds a multi-edge clock, it passes to the Quartus II software and issues an error message.
- Clocks in the same clock -`domain` are annotated with the `create_relative_clock` command to create related clocks in the Quartus II software.
- The Quartus II software assumes the first clock edge to be at time 0.0. If the Precision RTL Synthesis software waveform has a first transition at a time different than time zero (0.0), the Precision RTL 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 RTL 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 RTL 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 RTL 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 RTL Synthesis software allows you to set output delays on pins inside the design, these constraints are not sent to the Quartus II software, and a message is displayed.

set_false_path

The false path constraint is specified in the Precision RTL Synthesis software as shown in Example 11–6.

Example 11–6. 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 "?."

This setting in the Precision RTL Synthesis software is mapped to a `set_timing_cut_assignment` setting in the Quartus II software.

The node lists for this assignment represents top-level ports and/or nets connected to instances (end points of timing assignments). The node lists can contain wildcards. The Quartus II software does not support bus notation such as A[7:4] in the node lists.

The Quartus II software does not support any `setup`, `hold`, `rise`, or `fall` options for this assignment.

The Quartus II software does not support false paths with the `through` path specification. Any setting in the Precision RTL Synthesis software with a `-through` specification cannot be mapped to a setting in the Quartus II software.

If you use the `from` or `to` option without using both options, the Precision RTL Synthesis command is converted to a Quartus II command using wildcards. Table 11–3 lists these `set_false_path` constraints in the Precision RTL Synthesis software and the Quartus II software equivalent.

Table 11–3. set_false_path Constraints

Precision RTL 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 RTL Synthesis software as shown in Example 11–7.

Example 11–7. 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 lists 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` is to indicate that source clock cycles should be considered for the multiplier. The option `end` is to indicate that destination clock cycles should be considered for the multiplier. The default is to reference the end clock.

This setting in Precision RTL Synthesis software is mapped to a `set_multicycle_assignment` setting in the Quartus II software.

The node lists represent top-level ports and/or nets connected to instances (end points of timing assignments). The node lists can contain wildcards; the Quartus II software automatically expands all wildcards. The Quartus II software does not support bus notation as A[7:4] in the node list.

If you use the `from` or `to` option without using both options, the Precision RTL Synthesis command is converted to a Quartus II command using wildcards. Table 11–4 lists the `set_multicycle_path` constraints in the Precision RTL Synthesis software and the Quartus II software equivalent

Table 11–4. set_multicycle_path Constraints

Precision RTL 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 does not support the `rise` or `fall` options on this assignment.

The Quartus II software does not support multicycle path with a `through` path specification. Any setting in Precision RTL Synthesis software with a `-through` specification cannot be mapped to a setting in the Quartus II software.

Megafunctions & Architecture-Specific Features

Altera provides parameterizable megafunctions including LPM, device-specific Altera megafunctions, intellectual property (IP) available as Altera MegaCore functions, and IP available through the Altera Megafunction Partners Program (AMPP^{SM}). You can use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code.

For more details about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, consult the appropriate IP documentation.

If you want to instantiate a megafunction in your HDL code, you can use the MegaWizard Plug-In Manager to parameterize the function or you can instantiate the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface for customizing and parameterizing any available megafunction for the design. The "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" section describes the MegaWizard flow with the Precision RTL Synthesis software.

The Precision RTL Synthesis software automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction will provide optimal results. The Precision RTL Synthesis

software also provides options to control inference of certain types of megafunctions, as described in the "Inferring Altera Megafunctions from HDL Code" section.

For a detailed information 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. In addition, the chapter provides coding style recommendations and examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction and to create a custom megafunction variation, the MegaWizard creates either a VHDL or Verilog HDL wrapper file. This file instantiates the megafunction (a black-box methodology) or, for some megafunctions, generates a fully synthesizeable netlist for improved results using EDA synthesis tools such as the Precision RTL Synthesis software (a clear-box methodology).

Clear-Box Methodology

You can use the MegaWizard Plug-In Manager to generate a fully synthesizable netlist. This flow is referred to as a clear-box methodology because the Precision RTL Synthesis software can "see" into the megafunction file. The clear box feature enables the synthesis tool to report more accurate resource utilization and timing estimates, taking better advantage of timing driven optimization.

This clear-box feature of the MegaWizard Plug-In Manager is turned on by choosing the **Generate clear box body (for EDA tools only)** in the **MegaWizard Plug-In Manager** for certain megafunctions. If the option does not appear, then clear box models are not supported for the selected megafunction. Turning on this option causes the MegaWizard Plug-In Manager to generate a synthesizable clear box netlist instead of the megafunction wrapper file described in the "Black-Box Methodology" section.

Using MegaWizard-Generated Verilog HDL Files for Clear Box Megafunction Instantiation

The MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file <*output*>**_inst.v** for use in your Precision RTL Synthesis design. This file can help you instantiate the megafunction clear box netlist file, <*output file*>**.v**, in your top-level design. Include the megafunction clear box netlist file in your Precision RTL Synthesis project and the information gets passed to the Quartus II software in the Precision RTL Synthesis-generated EDIF output file.

Using MegaWizard-Generated VHDL Files for Clear Box 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 design. These files help to instantiate the megafunction clear box netlist file, <***output file***>.**vhd**, in your top-level design. Include the megafunction clear box netlist file in your Precision RTL Synthesis project and the information gets passed to the Quartus II software in the Precision RTL Synthesis-generated EDIF output file.

Black-Box Methodology

Using the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black-box methodology because the megafunction is treated as a black box in the Precision RTL Synthesis software. The black-box wrapper file is generated by default in the MegaWizard Plug-In Manager and is available for all megafunctions.

The black-box methodology does not allow the synthesis tool any visibility into the function module and so does not take full advantage of the synthesis tool's timing driven optimization.

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Black-Box 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 RTL Synthesis design. The instantiation template file helps to instantiate the megafunction variation wrapper file, <***output file***>**.v**, in your top-level design. Add the hollow-body black-box module declaration <***output file***>**_bb.v** to your Precision RTL Synthesis project to describe the port connections of the black box.

You do not have to include the megafunction variation wrapper file <*output file*>**.v** in your Precision RTL Synthesis project, but you must add it to your Quartus II project along with your Precision RTL synthesis-generated EDIF netlist. Alternately, you can include the file in your Precision project and then right-click on the file in the input file list, and select **Properties**. In the input file properties dialog, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision RTL Synthesis software does not compile this file and the tool makes a copy of the file in the appropriate directory so that the Quartus II software can compile the design during placement and routing.

Using MegaWizard Plug-In Manager-Generated VHDL Files for Black-Box 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 RTL Synthesis design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>.**vhd**, in your top-level design.

You do not have to include the megafunction variation wrapper file, <*output file*>.**vhd**, in your Precision RTL synthesis project, but you must add it to your Quartus II project with your Precision RTL synthesis-generated EDIF netlist. Alternately, you can include the file in your Precision project and then right-click on the file in the input file list, and select **Properties**. In the input file properties dialog, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision RTL Synthesis software does not compile this file and the tool makes a copy of the file in the appropriate directory so that the Quartus II software can compile the design during placement and routing.

Inferring Altera Megafunctions from HDL Code

The Precision RTL Synthesis software automatically recognizes certain types of HDL code and maps arithmetic and relational operators, and memory (RAM and ROM), to efficient technology-specific implementations. This allows for the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when a megafunction will provide optimal results. In some cases, the Precision RTL Synthesis software has options that you can use to disable or control inference.

For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision RTL Synthesis Manuals Bookcase in the Help menu.

Multipliers

The Precision RTL 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 RTL 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 RTL Synthesis software uses DSP blocks available in the Stratix series of devices to implement multipliers. The default setting is **AUTO**, to allow Precision RTL 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 RTL 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 RTL Synthesis software are shown in Table 11–5.

Table 11–5. Options for DEDICATED_MULT Parameter to Control Multiplier Implementation in Precision RTL 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

Perform the following steps to set the **Use Dedicated Multiplier** option in the Precision RTL Synthesis GUI:

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 11–8 and Example 11–9.

Example 11–8. Setting the dedicated_mult Attribute in Verilog HDL

```
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 11–9. 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 dedicated_mult attribute is set may be synthesized away by the Precision RTL 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 11–10.

Example 11–10. Setting the preserve_signal Attribute in Verilog HDL

```
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 11–11. Setting the preserve_signal Attribute in VHDL

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 11–12 and Example 11–13 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 11–12. 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 11–13. 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 & Multiplier-Adders

The Precision RTL Synthesis software detects multiply-accumulators or multiply-adders in HDL code and infers an `altmult_accum` or `altmult_add` megafunction so that the logic can be placed in DSP blocks, or maps directly to device atoms to implement the multiplier in the appropriate type of logic.

The Precision RTL Synthesis software supports inference for these functions only if the target device family has dedicated DSP blocks.

The Precision RTL Synthesis software also allows you to control the device resources used to implement multiply-accumulators or multiply-adders in your project or in a particular module. Refer to the **"Controlling DSP Block Inference" on page 11–26** section 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 web site.

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 RTL Synthesis Manuals Bookcase in the Help menu.

Controlling DSP Block Inference

By default the Precision RTL 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 `altumult_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 11–14 and Example 11–15.

Example 11–14. Setting the extract_mac Attribute in Verilog HDL

```
//synthesis attribute <module name> extract_mac <value>
```

Example 11–15. 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 as described in the "Controlling DSP Block Inference" section. See that section for syntax details.

Example 11–16 and Example 11–17 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 11–16. Use of extract_mac, dedicated_mult & 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 11–17. Use of 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 & ROM

The Precision RTL 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 RTL Synthesis Manuals Bookcase in the Help menu.

Incremental Compilation & 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 part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations can be made dramatically faster by focusing new compilations on particular design partitions and merging results with the results of previous compilations of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual blocks and then integrate them into a final design and optimize it at the top level.

Using the Precision RTL Synthesis software, you can create different netlist files for different partitions of a design hierarchy. Doing this makes each partition independent of the others for either a top-down or a bottom-up incremental compilation or LogicLock design flow. In either case, only the portions of a design that have been updated must be recompiled during design iterations. You can make changes and resynthesize one partition in a design to create a new netlist without affecting the synthesis results or fitting of other partitions. The following steps show the general top-down compilation flow when using these features of the Quartus II software:

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.
3. Create a project with multiple implementations (or create multiple projects) in the Precision RTL Synthesis software, one for each partition in the design.
4. Disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and synthesize each implementation or each project in the Precision RTL Synthesis software, and make constraints as in the regular design flow.
6. Import the EDIF netlist and the Tcl file for each partition into the Quartus II software and set up the Quartus II project(s) to use the incremental compilation or LogicLock methodology.
7. Compile your design in the Quartus II software and preserve the compilation results using the post-fit netlist type in incremental compilation or back-annotation in the LogicLock methodology.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate the new EDIF netlist and Tcl file. Do not resynthesize the implementations or projects for the unchanged partitions.
9. Import the new EDIF netlist and Tcl file into the Quartus II software and recompile the design in the Quartus II software using the incremental compilation or LogicLock methodology.

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 & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about using the LogicLock feature in the Quartus II software, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

Hierarchy & 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 RTL 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 *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design with Separate Netlist Files

The first step in 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 that you can take advantage of the incremental compilation and LogicLock design flows 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 RTL project, which allows you to switch between partitions without leaving the current project file, or you can create a separate project for each partition if you need separate projects for a bottom-up or team-based design flow.

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* in the Precision Manuals Bookcase in the Help menu.

When synthesizing the implementations for lower level modules, perform these steps:

1. Turn off **Add IO Pads** on the **Optimization** page under **Set Options** (Tools menu).
2. Read the HDL files for the modules.

   Modules may include black-box instantiations of lower level modules that are also maintained as separate EDIF files.

3. Add constraints for all partitions in the design.

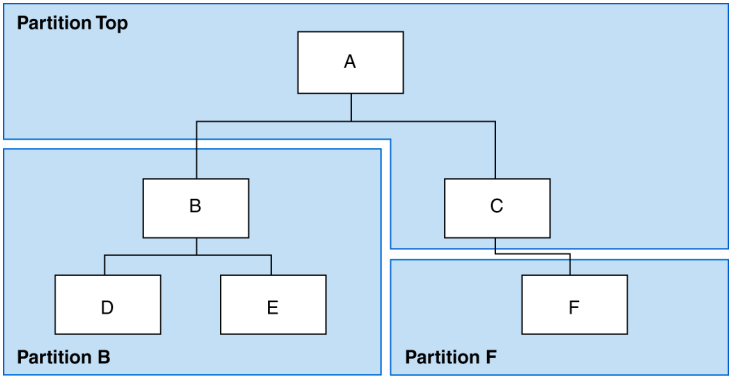
When synthesizing the top-level design implementation, perform these steps:

1. Read the HDL files for top-level designs.
2. Create black boxes for lower level modules in the top-level design.
3. Add constraints.

In a top-down incremental compilation flow, 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 RTL Synthesis project, or in the Quartus II project.

The following sections describe an example of implementing black boxes to create separate EDIF netlists. Figure 11–2 shows an example of a design hierarchy separated into various partitions.

Figure 11–2. Partitions in a Hierarchical Design



In Figure 11–2, the top-level partition contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in the C subblock. 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 B and its submodules D and E, while a third netlist is created for 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.

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 example for top-level file **A.v** follows. 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 11–18. 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 need 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 the following example. 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 11–19. 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 in the incremental compilation or LogicLock design methodologies in the Quartus II software.

Creating Quartus II Projects for Multiple EDIF Files

The Precision RTL Synthesis software creates a Tcl file for each EDIF file, and 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 Precision RTL 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 RTL Synthesis-Generated Tcl Script" on page 11–14*.

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 and floorplan location assignments for each partition in the design within a single Quartus II project. This methodology provides the best quality of results and performance preservation during incremental changes to your design. You may need 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. Alternately, you can use the LogicLock design methodology to import each lower-level partition and maintain placement results.

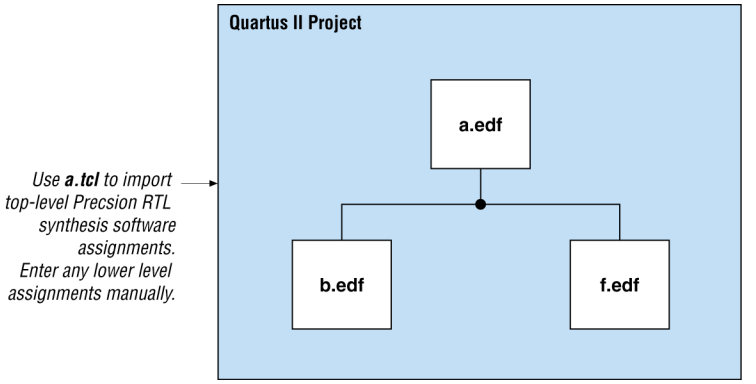
The following sections describe how to create the Quartus II projects for these two design flows.

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.

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 RTL Synthesis software are not forward-annotated. Enter these constraints manually in your Quartus II project.

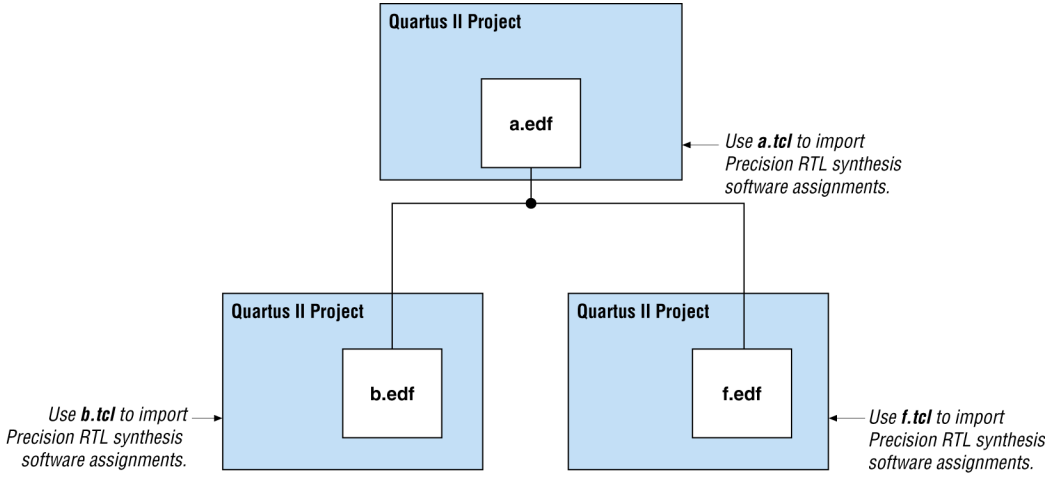
Figure 11–3. Design Flow Using Multiple EDIF Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up Flow

Use the Tcl files generated by the Precision RTL Synthesis software for each Precision RTL 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 the incremental compilation or LogicLock design methodology. 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. Figures 11–4 shows the design flow for the example design in Figure 11–2.

Figure 11–4. Design Flow: Using Multiple EDIF Files with Multiple Quartus II Projects



Conclusion

Advanced synthesis is an important part of the design flow. The Mentor Graphics Precision RTL Synthesis software and Quartus II design flow allows you to control how to prepare your design files for the Quartus II place-and-route process. This allows you to improve performance and optimize a design for use with Altera devices. Several of the methodologies outlined in this chapter can help you optimize a design to achieve performance goals and save design time.

Document Revision History

Table 11–7 shows the revision history for this document.

Table 11–7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 6.1.0	Added revision history to the document.	
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 9 was formerly Chapter 10 in version 5.0.	
May 2005 v5.0.0	Chapter 10 was formerly chapter 8 in version 4.2.	
Dec. 2004 v2.1	• Chapter 9 was formerly Chapter 10 in version 4.1. • Updates to tables and figures. • New functionality for Quartus II software version 4.2.	
June 2004 v2.0	• Updates to tables and figures. • New functionality for Quartus II software version 4.1.	
Feb. 2004 v1.0	Initial release.	

12. Synopsys Design Compiler FPGA Support

QII51014-6.1.0

Introduction

Programmable logic device (PLD) designs have reached the complexity and performance requirements of ASIC designs. As a result, advanced synthesis has taken on a more important role in the design process. This chapter documents the usage and design flow of the Synopsys Design Compiler FPGA (DC FPGA) synthesis software with Altera® devices and Quartus® II software. DC FPGA supports Stratix® II, Stratix, Stratix GX, Cyclone™ II, and Cyclone devices.

This chapter assumes that you have set up and licensed the DC FPGA software and Altera Quartus II software.

This chapter is primarily intended for ASIC designers experienced with the Design Compiler (DC) software who are now developing PLD designs, and experienced PLD designers who would like an introduction to the Synopsys DC FPGA software.

To obtain the DC FPGA software, libraries, and instructions on general product usage, go to the Synopsys web site at **http://solvnet.synopsys.com/retrieve/012889.html**

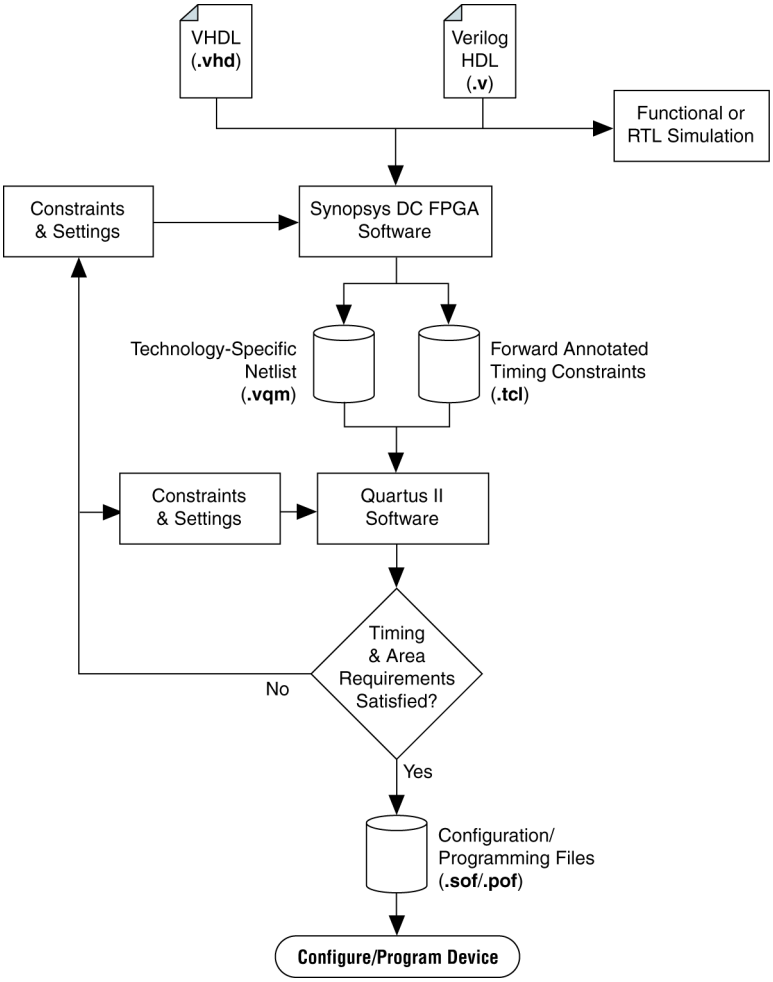
The following areas are covered in this chapter:

- General design flow with the DC FPGA software and the Quartus II software
- Initialization procedure using the **.synopsys_dc.setup** file for targeting Altera devices
- Using Altera megafunctions with the DC FPGA software
- Reading design files into the DC FPGA software
- Applying synthesis and timing constraints
- Reporting and saving design information
- Exporting designs to the Quartus II software

Design Flow Using the DC FPGA Software & the Quartus II Software

A high-level overview of the recommended design flow for using the DC FPGA software with the Quartus II software is shown in Figure 12–1.

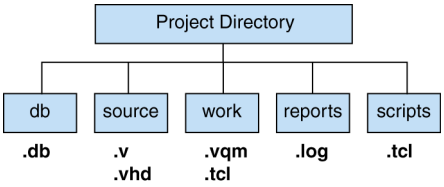
Figure 12–1. Design Flow Using the DC FPGA Software & the Quartus II Software



Setup of the DC FPGA Software Environment for Altera Device Families

Altera recommends that you organize your project directory with several subdirectories. A recommended project hierarchy is shown in Figure 12–2.

Figure 12–2. Project Hierarchy



To use the DC FPGA software to synthesize HDL designs for use with the Quartus II software, the required settings should be included in your **.synopsys_dc.setup** initialization file. This file is used to define global variables and direct the DC FPGA software to the proper libraries used for synthesis, as well as set internal assignments for synthesizing designs for Altera devices.

The **.synopsys_dc.setup** file can reside in any one of three locations and be read by the DC FPGA software. The DC FPGA software automatically reads the **.synopsys_dc.setup** file at startup in the following order of precedence:

1. Current directory where you run the DC FPGA software shell.
2. Home directory.
3. The DC FPGA software installation directory.

The DC FPGA software has vendor-specific setup files for each of the Altera logic families in the installation directory. These vendor-specific setup files are found where you have installed the libraries (*<dcfpga_rootdir>***/libraries/fpga/altera**) and are named in the form **synopsys_dc_***<logic family>***.setup**. For example, if you want to use the default setup for synthesizing an Altera Stratix device, you must link to or copy the **synopsys_dc_stratix.setup** to your home or current directory and rename the file **.synopsys_dc.setup**.

Synopsys recommends using the vendor-specific setup files provided with each release of the DC FPGA software to ensure that you have all the correct settings and obtain the best quality results.

Example 12–1 contains the recommended synthesis settings for the Stratix II device architecture.

Example 12–1. Recommended Synthesis Settings for Stratix II Device Architecture

```
# Setup file for Altera Stratixii
# TCL style setup file but will work for original DC shell as well
# Need to define the root location of the libraries by chaning the variable
$dcfpga_lib_path

set dcfpga_lib_path "<dcfpga_rootdir>/libraries/fpga/altera"


set search_path ". $dcfpga_lib_path $dcfpga_lib_path/STRATIXII $search_path"
set target_library "stratixii.db"
set synthetic_library "tmg.sldb altera_mf.sldb lpm.sldb"
set link_library "* stratixii.db tmg.sldb altera_mf.sldb lpm.sldb stratixii_mf.sldb"

set_fpga_defaults altera_stratixii
```

After generating your **.synopsys_dc.setup** file, run the DC FPGA software in either the Tcl shell or in the Design Compiler software shell without Tcl support. Run the DC FPGA software shell at a command prompt by typing `fpga_shell-t` or `fpga_shell -tcl` for the Tcl shell version of the DC FPGA software. Run the non-Tcl version of the DC FPGA software with the `fpga_shell` command. Altera recommends using the Tcl shell for all of your synthesis work.

If you have created a Tcl synthesis script for use in the DC FPGA software and wish to run it immediately at startup, you can start the DC FPGA software shell and run the script with the command shown in the example below:

`fpga_shell-t -f` *<path>/<script filename>*`.tcl` ↵

Otherwise, you can run your scripts at any time at the `fpga_shell-t>` prompt with the `source` command. An example is shown below:

`source` *<path>/<script filename>*`.tcl` ↵

Megafunctions & Architecture-Specific Features

Altera provides parameterized megafunctions including library of parameterized modules (LPMs), device-specific Altera megafunctions, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP). You can use megafunctions by instantiating them in your HDL code, or by inferring them from your HDL code during synthesis in the DC FPGA software.

For more details on specific Altera megafunctions, refer to the Quartus II Help.

The DC FPGA software automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. The DC FPGA software also provides options to control inference of certain types of megafunctions, as described in the section "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 12–6.

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. In addition, the chapter provides coding style recommendations and examples for inferring megafunctions in Altera devices.

If you instantiate a megafunction in your HDL code, you can use the MegaWizard Plug-In Manager to parameterize the function, or you can instantiate the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface in the Quartus II software for customizing and parameterizing megafunctions. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 12–6 describes the MegaWizard Plug-In Manager flow with the DC FPGA synthesis software.

Instantiating 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 that instantiates the megafunction (a black box methodology). The MegaWizard can also generate a fully elaborated netlist that is read by EDA synthesis tools, such as the DC FPGA (a clear box methodology). Both clear box and black box methodologies are described in the following sections.

Clear Box Methodology

You can use the MegaWizard Plug-In Manager to generate a fully synthesizeable netlist. This flow is referred to as a clear box methodology because starting in V-2005.06, the DC FPGA software can look into the megafunction file. The clear box feature enables the synthesis tool to report more accurate timing estimates and resource utilization, while taking a better advantage of timing driven optimization than a black box methodology.

This clear box feature is enabled by turning on the **Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** option in the MegaWizard Plug-In Manager for certain megafunctions. DC FPGA supports clear box megafunctions for `altmult_add`, `almult_accum`, `altsyncram` and `altshift_taps`. If the option does not appear, then clear box models are not supported for the selected megafunction.

The library declarations in the MegaWizard generated VHDL output files need to be manually commented out to work properly with the DC FPGA.

Reading Megafunction Wizard-Generated Synthesizable Clear Box Netlist Files for Megafunction Instantiation

The DC FPGA software analyzes and elaborates the Megafunction Wizard-generated Verilog HDL <*output file*>**.v** or VHDL <*output file*>**.vhd** netlist that contains the parameters needed by the Quartus II software to properly configure and instantiate your megafunction. Analyze the clear box netlist files along with the rest of the RTL files during synthesis in DC FPGA. The resulting netlist contains all the primitives that are part of the clear box netlist. There is no need to put the clear box netlist file in your Quartus II project along with your DC FPGA generated netlist file.

Using the clear box Megafunction Wizard-generated netlist files provides the DC FPGA software an understanding of their timing arcs and resource usage. The DC FPGA software uses timing information to optimize the surrounding circuits and resource data to better manage the overall resource usage for the whole design. The DC FPGA software takes the clear box netlist timing and area data into account when reporting the timing and resource utilization for the device.

Advanced Clear Box Support for the Direct-Instantiated or Inferred Clear Box Megafunctions

The DC FPGA provides advanced clear box support that enables a clear box implementation for the direct-instantiated or inferred megafunctions in your design. This methodology allows the DC FPGA to obtain the most accurate interface timing and area data for the megafunctions. Therefore, synthesis optimization is more effective, and timing and area reports are more accurate.

The following describes the setup and usage model for this advanced clear box support.

Design Compiler FPGA Setup

The advanced clear box flow will be enabled in the DC FPGA only when the **clearbox.sldb** synthetic library is added to the `synthetic_library` variable. For example:

```
set synthetic_library [concat clearbox.sldb $synthetic_library]
set link_library [concat clearbox.sldb $link_library]
```

Specify the path to the clear box loader (executable) in one of the following ways:

- Set the `synlib_cbx_exec_path` variable to the absolute path of the clear box loader before the compile command:
  `set synlib_cbx_exec_path` <*Quartus II installation directory /bin/clearbox*>
- Set the UNIX environment variable `CLEARBOX_EXEC_PATH` to the absolute path of the clear box loader. For example:
  `setenv CLEARBOX_EXEC_PATH` <*Quartus II installation directory /bin/clearbox*>

By default, the advance clear box flow is turned off. To enable the clear box advanced flow, add the following to your DC FPGA script. Set it before the compile command:

```
set fpga_altera_clearbox_for_user_cells true
```

UNIX Environment Setting

For the DC FPGA to work with the clear box loader, the following setting is necessary for the `LD_LIBRARY_PATH` environment variable. Assume the `QuartusII_Path` used below is set to the Quartus II installation directory.

On a Linux platform:

```
setenv LD_LIBRARY_PATH QuartusII_Path/linux:$LD_LIBRARY_PATH
```

On a Solaris platform:

```
setenv LD_LIBRARY_PATH QuartusII_Path/solaris:$LD_LIBRARY_PATH
```

Error Message

The only error message that you might encounter when trying to enable the advanced clear box flow is: `DCFPGA_UEGI-1`

The DC FPGA reports this error when one of the following situations occurs:

- It cannot find the clear box loader path. For example, the defined path is incorrect.
- The Loader is not found in the specified path.
- The Loader specified is not executable.

Sample Design Compiler FPGA Clear Box Setup Script

The TCL script shown in Example 12–2 is a DC FPGA clear box setup script. Use it before compiling the design in DC FPGA.

Example 12–2. Sample Clear Box Setup Script

```
set QuartusII_Path /tools/altera/qii51
set_unix_variable CLEARBOX_EXEC_PATH $QuartusII_Path/bin/clearbox
set old_llp [get_unix_variable LD_LIBRARY_PATH]
set platform [sh uname]

if { $platform == "Linux" } {
    set_unix_variable LD_LIBRARY_PATH $QuartusII_Path/linux: old_llp
} else {
    # Assume, if not linux, it is solaris
    set_unix_variable LD_LIBRARY_PATH $QuartusII_Path/solaris: old_llp

set synthetic_library [concat clearbox.sldb $synthetic_library]
set link_library [concat clearbox.sldb $link_library]

set fpga_altera_clearbox_for_user_cells true
```

Black Box Methodology

Using the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black box methodology because the megafunction is treated as a black box in the DCFPGA software. The black box wrapper file is generated by default in the MegaWizard Plug-In Manager and is available for all megafunctions. The black box methodology does not allow the synthesis tool any visibility into the function module and therefore, does not take full advantage of the synthesis tool's timing driven optimization.

There are two ways of instantiating Megafunction Wizard-generated functions in your design hierarchy loaded in the DC FPGA software. You can instantiate and compile the Verilog HDL or VHDL variation wrapper file description of your megafunction in the DC FPGA software, or you can instantiate a black box that just describes the ports of your megafunction variation wrapper file.

The library declarations in the MegaWizard generated VHDL output files need to be manually commented out to work properly with the DC FPGA.

Reading Megafunction Wizard-generated Variation Wrapper Files

The DC FPGA software has the ability to analyze and elaborate the Megafunction Wizard-generated Verilog HDL <*output file*>**.v** or VHDL <*output file*>**.vhd** netlist that contains the parameters needed by the Quartus II software to properly configure and instantiate your megafunction. The DC FPGA software may take advantage of this variation wrapper file during the optimization of your design to reduce area utilization and improve path delays. DC FPGA also supports altpll in a non-black box flow (that is, the DC FPGA can automatically derive PLL output clocks when the user has specified only the PLL input clock).

Using the megafunction variation wrapper file <*output file*>**.v** or <*output file*>**.vhd** in the DC FPGA software synthesis provides good synthesis results for area estimates, but actual timing results are best predicted after place-and-route inside the Quartus II software. However, reading the megafunction variation wrapper allows the DC FPGA software to provide better synthesis estimates over a black box methodology.

Using Megafunction Wizard-Generated Variation Wrapper Files in a Black Box Methodology

Instantiating the megafunction wizard-generated wrapper file without reading it in the DC FPGA software is referred to as a black box methodology because the megafunction is treated as an unknown container in the DC FPGA software.

The black box methodology does not allow synthesis software to have any visibility into the module, thereby not taking full advantage of the timing driven optimization of the DC FPGA software and preventing the software from estimating logic resources for the black box design.

Using Megafunction Wizard-Generated Verilog HDL Files for Black Box Megafunction Instantiation

By default, the MegaWizard Plug-In Manager generates the Verilog HDL instantiation template file <*output file*>**_inst.v** and the black box module declaration <*output_file*>**_bb.v** for use in your design in the DC FPGA software. The instantiation template file helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level design. Do not include the megafunction variation wrapper file in the DC FPGA software project if you are following the black box methodology. Instead, add the wrapper file and your generated Verilog Quartus Mapping (**.vqm**) netlist in your Quartus II project. Add the hollow body black box module declaration <*output file*>**_bb.v** to your linked design files in the DC FPGA software to describe the port connections of the black box.

Using Megafunction Wizard-Generated VHDL Files for Black Box Megafunction Instantiation

By default, 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 design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>**.vhd**, in your top-level design. Do not include the megafunction variation wrapper file in the DC FPGA software project. Instead, add the wrapper file and your generated Verilog Quartus Mapping netlist in your Quartus II project.

The DC FPGA software supports direct instantiation of all LPMs and megafunctions. For a complete list of all LPMs and Megafunctions, refer to the following two files in your Quartus II installation directory:

- <Quartus II installation directory> **/libraries/vhdl/lpm/lpm_pack.vhd**
- <Quartus II installation directory> **/libraries/vhdl/altera_mf/altera_mf_components.vhd**

DC FPGA supports direct instantiation of LPMs and megafunctions only. These macro functions include all Altera IP cores and all components listed in:

<***Quartus II installation directory***>**/libraries/vhdl/altera_mf_components.vhd** or **stratixgx_mf_components.vhd**.

The following example is the usage model using the **mypll** for direct instantiation:

1. During synthesis in DC FPGA, analyze the variation file **mypll.[v|vhd]** along with the rest of the RTL files.

2. During place-and-route in the Quartus II software, simply run the self-contained Verilog Quartus Mapping File. You do not need to put the variation file in the Verilog Quartus Mapping directory.

The benefit of using the direct instantiation method is that the DC FPGA is able to utilize the available clock enable pins of the LPMs and megafunctions during the automatic gated-clock conversion process.

Inferring Altera Megafunctions from HDL Code

The DC FPGA software automatically recognizes certain types of HDL code, and maps digital signal processing (DSP) functions and memory (RAM and ROM) to efficient, technology-specific implementations. This allows the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when it provides optimal results.

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*.

Depending on the coding style, if you do not adhere to these recommended HDL coding style guidelines, it is possible that the DC FPGA software and Quartus II software will not take advantage of the high performance DSP blocks and RAMs, and may instead

implement your logic using regular logic elements (LEs). This causes your logic to consume more area in your device and may adversely affect your design performance. Altera device families do not all share the same resources, so your HDL coding style may cause your logic to be implemented differently in each family. For example, Stratix devices contain dedicated DSP blocks which Cyclone devices lack. In a Cyclone device, multipliers are implemented in LEs.

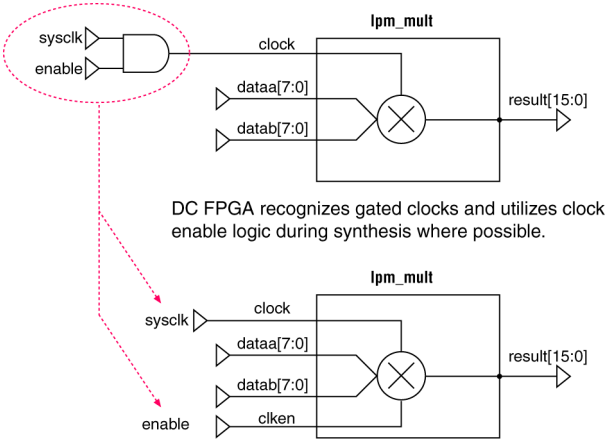
Example 12–3 shows Verilog HDL code that infers a two-port RAM that can be synthesized into an M512 RAM block of a Stratix device.

Example 12–3. Verilog HDL Code Inferring a Two-Port RAM

```
module example_ram (clk, we, rd_addr, wr_addr, data_in, data_out);
input clk, we;
input [15:0] data_in;
output [15:0] data_out;
input [7:0] rd_addr;
input [7:0] wr_addr;
reg [15:0] ram_data [7:0];
reg [15:0] data_out_reg;
always @ (posedge clk)
begin
if (we)
        ram_data[wr_addr] <= data_in;
data_out_reg <= ram_data[rd_addr];
end
assign data_out = data_out_reg;
endmodule
```

One of the strengths of the DC FPGA software is its gated clock conversion feature. Inferring megafunctions in HDL takes advantage of this feature. For gated clocks or clock enables designed outside of LPMs, Altera-specific megafunctions, and registers, the DC FPGA software merges the gated clock functions into these design elements using dedicated clock enable functionality during synthesis. The DC FPGA software reconfigures the megafunction block or register to synthesize the clock enable control logic. This can save area in your design and improve your design performance by reducing the gated clock path delay and the amount of logic used to implement the design. An illustration of this kind of gated clock optimization is shown in Figure 12–3.

Figure 12–3. Gated Clock Optimization



The DC FPGA software does not perform gated clock optimization on instantiated black box megafunctions or on instantiated megafunction variation wrapper file. The DC FPGA software performs gated clock optimization only on synthesizable inferred megafunctions.

Reading Design Files into the DC FPGA Software

The process of reading design files into the DC FPGA software is a two-step process where the DC FPGA software analyzes your HDL design for syntax errors, then elaborates the specified design. The elaboration process finds analyzed designs and instantiates them in the elaborated design's hierarchy. You must identify which supported language the files are written in when reading designs into the DC FPGA software. The supported HDL languages are listed in Table 12–1.

Table 12–1. Supported Design File Formats

Format	Description	Keyword	Extension
Verilog HDL (Synopsys Presto HDL)	Verilog hardware description language	verilog	.v
VHDL	VHSIC hardware description language	vhdl	.vhd
.db	Synopsys internal database format *(1)*	db	.db
EDIF	Electronic design interchange format	edif	.edf

Note to Table 12–1:
(1) The Design Compiler DB format file requires additional license keys.

To set most of the required synthesis settings to generate an optimal netlist, use the following command:

```
set_fpga_defaults <architecture_name>
```

For example:

```
set_fpga_defaults altera_stratixii
```

Use the following commands to analyze and elaborate HDL designs in the DC FPGA software:

```
analyze -f <verilog|vhdl> <design file> ↵
```

```
elaborate <design name> ↵
```

Once a design is analyzed, it is stored in a Synopsys library format file in your working directory for reuse. You need to re-analyze the design only when you change the source HDL file. Elaboration is performed after you have analyzed all of the subdesigns below your current design.

Another way to read your design is by using the `read_file` command. This can be used to read in gate-level netlists that are already mapped to a specific technology. The `read_file` command performs analysis and elaboration on Verilog HDL and VHDL designs that are written in register transfer level (RTL) format. The difference between the `read_file` command and the analyze and elaborate combination is that the `read_file` command elaborates every design read, which is unnecessary. Only the top-level design must be elaborated. The `read_file` command is useful if you have a previously synthesized block of logic that you want to re-use in your design.

To use the `read_file` command for a specific language, type the following command:

```
read_file -f <verilog|vhdl|db|edif> <design file> ↵
```

You can also read files in specific languages using the `read_verilog`, `read_vhdl`, `read_db`, and `read_edif` commands.

Once you have read all of your design files, specify the design you want to focus your work on with the `current_design` command. This is usually the top module or entity in your design that you wish to compile up to. To use this command, type the following:

```
current_design <design name> ↵
```

You then need to build your design from all of the analyzed HDL files with the link command. To use this command, type the following:

```
link ↵
```

After linking your designs successfully in the DC FPGA software, you should specify the constraints you are applying to your design. In the DC FPGA software, you have the capability of loading multiple levels of hierarchy and synthesizing specific blocks in a bottom-up synthesis methodology, or you can synthesize the entire design from the top-level module in a top-down synthesis methodology.

You can switch the current focus of the DC FPGA software between the designs loaded by using the `current_design` command. This changes your current focus onto the design specified, and all subsequent constraints and commands will apply to that design.

If you have read Quartus II megafunction wizard-generated designs or third-party IP into the DC FPGA software, you can instruct the DC FPGA software not to synthesize the IP. Use the `set_dont_touch` constraint and apply it to each module of your design that you do not want synthesized. To use this command, type the following:

```
set_dont_touch <design name> ↵
```

Using the `set_dont_touch` command can be helpful in a bottom-up synthesis methodology, where you optimize designs at the lower levels of your hierarchy first and do not allow the DC FPGA software to resynthesize them later during the top-level integration. However, depending on the design's HDL coding, you might want to allow top-level resynthesis to get further area reduction and improved path delays. For best results, Altera recommends following the top-down synthesis methodology and not using the `set_dont_touch` command on lower level designs.

Selecting a Target Device

If you do not select an Altera device, the DC FPGA software, by default, synthesizes for the fastest speed grade of the logic family library that is loaded in your **.synopsys_dc.setup** file. If you are targeting a specific device of an Altera family, you must have the correct library linked, then specify the device for synthesis with the `set_fpga_target_device` command**.** To use this command, type the following:

```
set_fpga_target_device <device name> ↵
```

You can have the DC FPGA software produce a list of all available devices in the linked library by adding the `-show_all` option to the `set_fpga_target_device` command. An example of this list of devices for the Stratix II library is shown in Example 12–4.

Example 12–4. List of Available Devices in the Linked Library Using the -show_all Option

```
Loading db file '/dc_fpga/libraries/fpga/altera/STRATIXII/stratixii.db'

  Valid device names are:

Part             Pins     FFs    Speed Grades
-------------------------------------------------
AUTO *           0        0        FASTEST
EP2S15F484       484      12480    C4
EP2S15F672       672      12480    C4
EP2S30F484       484      27104    C4
EP2S30F672       672      27104    C4
EP2S60F484       484      48352    C4
EP2S60F672       672      48352    C4
EP2S60F1020      1020     48352    C4
EP2S90F1020      1020     72768    C4
EP2S90F1508      1508     72768    C4
EP2S130F1020     1020     106032   C4
EP2S130F1508     1508     106032   C4
EP2S180F1020     1020     143520   C4
EP2S180F1508     1508     143520   C4

  * Default part
```

For example, if you want to target the C4 speed grade device of the Stratix II EP2S60F672 device, apply the following constraint:

```
set_fpga_target_device EP2S60F672C4
```

Timing & Synthesis Constraints

You must create timing and synthesis constraints for your design for the DC FPGA software to optimize your design performance. The timing constraints specify your desired clocks and their characteristics, input and output delays, and timing exceptions such as false paths and multi-cycle paths. The synthesis constraints define the device, the type of I/O buffers that should be used for top-level ports, and the maximum register fan-out threshold before buffer insertion is performed. Synopsys Design Constraints (SDCs) are Tcl-format commands that are widely used in many EDA software applications. The DC FPGA software supports the same SDC commands that the full version of the Design Compiler software supports. However, certain constraints that are used in ASIC synthesis are not applicable to programmable logic synthesis, so the DC FPGA software ignores them.

The DC FPGA software supports the following constraints:

- `create_clock`
- `set_max_delay`
- `set_propagated_clock`
- `set_input_delay`
- `set_output_delay`
- `set_multicycle_path`
- `set_false_path`
- `set_disable_timing`
- `set_fpga_resource_limit`
- `set_register_max_fanout`
- `set_max_fanout`
- `set_fpga_target_device`

For the syntax and full usage of these commands, refer to the *Synopsys DC FPGA User Guide*.

For synthesis with the DC FPGA software, minimum timing analysis is not necessary, as it primarily looks at setup timing optimization to achieve the fastest clock frequency for your design. Altera recommends adding additional minimum timing constraints to your design inside the Quartus II software.

The DC FPGA forward annotates all the clock, timing exceptions, and I/O delay constraints to Quartus II when the `write_par_constraint` command is used in the DC FPGA. For more information about this command, refer to "Exporting Designs to the Quartus II Software" on page 12–22. Since the Quartus II software does not support the `through` option for the timing exception constraints, the DC FPGA does not forward annotate constraints that use the `through` option.

In the DC FPGA software, timing constraints applied to inferred RAM, ROM, shift registers, and DSP MAC functions are obeyed. However, these constraints are not forward-annotated to the Quartus II software because these functions are inferred to Altera megafunctions. The Quartus II software does not support timing constraints applied to megafunctions. The workaround is to run the Verilog Quartus Mapping/EDIF netlist through analysis and synthesis in the Quartus II software (**quartus_map**). All megafunctions expand to atom primitives. These atom primitives can be processed by the Quartus II software. You can then apply constraints to the internal atoms of the megafunctions.

The timing reports generated from the DC FPGA software are preliminary estimates of the path delays in your design, and accurate timing is reported only after place-and-route is performed with the Quartus II software.

The DC FPGA software also performs cross-hierarchical boundary optimization. Altera recommends running this command before a compilation:

```
ungroup -small 500 ↵
```

This allows the DC FPGA software to potentially improve area reduction and performance improvement by ungrouping smaller blocks of logic in your design hierarchy and combining functions.

Compilation & Synthesis

After applying timing and synthesis constraints, you can begin the compilation and synthesis process. The `compile` command runs this process within the DC FPGA software. To run a compilation, at the shell prompt type:

```
compile ↵
```

The compilation process performs two kinds of optimization:

- Architectural optimization focuses on the HDL description and performs high-level synthesis tasks such as sharing resources and sub-expressions, selecting Synopsys Design Ware implementations, and re-ordering operators.
- Gate-level optimization works on the generic netlist created by logic synthesis and works to improve the mapping efficiency to save area and improve performance by minimizing path delays.

Compilation can be done using a top-down synthesis methodology or a bottom-up synthesis methodology. The top-down synthesis methodology involves a single compilation of your entire design with the focus on the top module or entity of your design. The bottom-up synthesis methodology involves incremental compilation of major blocks in your design hierarchy and top-level integration and optimization. Either methodology can be applied when synthesizing for Altera devices. For best results, Altera recommends following the top-down synthesis methodology.

An example synthesis script that reads the design, applies timing constraints, reports results, saves the synthesized netlist file in the Verilog Quartus Mapping File format, and creates the Tcl scripts to work with the

Quartus II software is shown in Example 12–5. It uses the command `write_fpga`, which is described in "write_fpga Command" on page 12–22.

Example 12–5. Sample Synthesis Script

```
# Setup output directories
set outdir ./design
file delete -force $outdir
file mkdir $outdir
set rptdir ./report
file delete -force $rptdir
file mkdir $rptdir
#  Enable Presto compiler for VHDL design files
#  set hdlin_enable_presto_for_vhdl TRUE
# Setup libraries
define_design_lib work-path .$outdir/work
file mkdir $outdir/work
analyze -format verilog ./source/mult_box.v
analyze -format verilog ./source/mult_ram.v
analyze -format verilog ./source/top_module.v
elaborate top_module
link
current_design top_module
create_clock -period 5 [get_ports clk]
set_input_delay -max 2 -clock clk [get_ports {data_in_* mode_in}]
set_input_delay -min 0.5 -clock clk [get_ports {data_in_* mode_in}]
set_output_delay -max 2 -clock clk [get_ports {data_out ram_data_out_port} ]
set_output_delay -min 0.5 -clock clk [get_ports {data_out ram_data_out_port} ]
set_false_path -from [get_ports reset]
ungroup -small 500
compile
report_timing > $rptdir/top_module.log
report_fpga > $rptdir/top_module_fpga.log
write_fpga $outdir
quit
```

Reporting Design Information

After compilation is complete, the DC FPGA software reports information about your design. You can specify which kinds of reports you want generated with the reporting commands shown in Table 12–2.

Table 12–2. Reporting Commands

Object	Command	Description
Design	report_design	Reports design characteristics
	report_area	Reports design size and object counts
	report_hierarchy	Reports design hierarchy
	report_resources	Reports resource implementations
	report_fpga	Reports FPGA resource utilization statistics for the design
Instances	report_cell	Displays information about instances
References	report_reference	Displays information about references
Ports	report_port	Displays information about ports
	report_bus	Displays information about bused ports
Nets	report_net	Reports net characteristics
	report_bus	Reports bused net characteristics
Clocks	report_clock	Displays information about clocks
Timing	report_timing	Checks the timing of the design
	report_constraint	Checks the design constraints
	check_timing	Checks for unconstrained timing paths and clock-gating logic
	report_design	Shows operating conditions, timing ranges, internal input and output, and disabled timing arcs
	report_port	Shows unconstrained input and output ports and port loading
	report_timing_requirements	Shows all timing exceptions set on the design
	report_clock	Checks the clock definition and clock skew information
	derive_clocks	Checks internal clock and unused registers
	report_path_group	Shows all timing path groups in the design
Cell Attributes	get_cells	Shows all cell instances that have a specific attribute

For more information about the usage of these commands, refer to the *Synopsys DC FPGA User Guide*.

The DC FPGA software only provides preliminary estimates of your design's timing delays because the timing of your design cannot be accurately predicted until the Quartus II software has placed and routed your design.

Saving Synthesis Results

After synthesis, the technology-mapped design can be saved to a file in one of the following four formats: Verilog HDL, VHDL, Synopsys internal DB, or EDIF.

The Quartus II software accepts an EDIF netlist or Verilog Quartus Mapping netlist synthesized from the DC FPGA software. The default output netlist from the DC FPGA software is Verilog Quartus Mapping. The Verilog Quartus Mapping File format follows a subset of Verilog HDL rules. You can use the same Verilog Quartus Mapping netlist format with the Quartus II software and formal verification.

Use the `write` command to save your design work. The syntax for this command is shown in Example 12–6.

Example 12–6. Syntax Using the write Command

```
write -format <verilog|db|edif> -output <file name> <design list>
[-hierarchy] ↵
```

The `-hierarchy` option causes the DC FPGA software to write all the designs within the hierarchy of the current design. The DC FPGA default flow to interface with Quartus II software uses the Verilog Quartus Mapping netlist.

To generate a Verilog Quartus Mapping netlist, set the required settings using the commands shown in Example 12–7.

Example 12–7. Generating a Verilog Quartus Mapping Netlist

```
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names -rules verilog -hier
write -format verilog -hier -o <design_top>.vqm
```

The Synopsys internal DB format is useful when you have synthesized your design and want to reuse it later in the DC FPGA software. The DB file contains your constraints and synthesized design netlist, and loads into the DC FPGA software faster than Verilog HDL or VHDL designs.

You can also write out your design constraints in Tcl format for export to the Quartus II software with the `write_par_constraint` command or by using the write_fpga command. These commands are explained in "Exporting Designs to the Quartus II Software".

Exporting Designs to the Quartus II Software

The DC FPGA software can create two Tcl scripts that start the Quartus II software, create your initial design project, apply the exported timing constraints, and compile your design in the Quartus II software.

You can generate the two Tcl scripts by using write and `write_par_constraint` command together, or by using the `write_fpga` command alone.

write_fpga Command

The recommended method to export all of the place-and-route files from the DC FPGA software is to use the `write_fpga` command. This command is used after the compile. Example 12–8 shows how the `write_fpga` command is used.

Example 12–8. Using the write_fpga Command after Compile

```
compile
write_fpga <outputdir>
```

The `write _fpga` command will do the following in one step:

Example 12–9. Using the write_fpga Command to Generate All Files

```
write -hier -f db -o $outputdir/top_module.db
write -hier -f edif -o $outputdir/top_module.edf
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names -rules verilog -hier
write -format verilog -hier -o <design_top>.vqm
write_par_constraint $outputdir/top_module_quartus_setup.tcl
```

When you use the `write_fpga` command, it generates all files in the current work directory or in the directory you specify (entering an output directory is optional) and generates the output files based on the current design file name.

write & write_par_constraint Commands

The `write` command is used to generate a post synthesis netlist for place-and-route and formal verification. You should use a Verilog Quartus Mapping formatting netlist to work with the Quartus II software, beginning with the DC FPGA software, version 2005.09. Example 12–10 uses the write and write_par_constraint commands to generate the Verilog Quartus Mapping File and Tcl scripts:

Example 12–10. Using the write & write_par_constraint Commands

```
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names –rules verilog –hier
write –format verilog -hier -o <design_top>.vqm
```

Tcl scripts that start the Quartus II software and forward annotate the timing constraints can be generated using the write_par_constraint command.

`write_par_constraint` <*user-specified file name*>`.tcl` ↵

This command generates both Tcl scripts in one operation. The first Tcl script has the name you specify in the `write_par_constraint` command. This script creates and compiles your Quartus II project. The second script is automatically generated and named <*top_module*>_const**.tcl** by default and contains your exported timing constraints from the DC FPGA software. This constraint file is sourced by the <*user-specified file name*>**.tcl** script and applies the timing constraints used in the DC FPGA software to your project in the Quartus II software.

For example, if your design is called **dma_controller**, and you run the command, `write_par_constraint run_quartus.tcl`, the DC FPGA software produces two Tcl scripts called `run_quartus.tcl` and `dma_controller_const.tcl`.

Using Tcl Scripts with Quartus II Software

To use this Tcl script in the Quartus II Tcl shell, type the following command at a command prompt:

`quartus_sh -t` <*user-specified file name*>`.tcl` ↵

To run this Tcl script in the Quartus II software GUI, type the following command at the Quartus II Tcl console prompt:

`source` <*user-specified file name*>`.tcl` ↵

The ability to run scripts in the Tcl console is useful when performing an initial compilation of your design to view post place-and-route timing and device utilization results, but the advanced Quartus II options that control the compilation process are not available.

To create a Quartus II project without performing compilation automatically, remove these lines from the script:

```
load_package flow
execute_flow -compile
```

Example 12–11. An Example Script

```
###############################################################################
# Generated by DC FPGA X-2005.09 on Wed Aug 10 04:20:01 2005
#
# Description: This TCL script is generated by DC FPGA using
#              write_par_constraint command. It is used to create a new Quartus
#              II project, specify timing constraint assignments in Quartus II,
#              and run quartus_map, quartus_fit, quartus_tan, & quartus_asm.
#
# Usage:  To execute this TCL script in batch mode: quartus_sh -t turboTop.tcl
#         To execute this TCL script in Quartus II GUI: source turboTop.tcl
#
#
#************    WARNING    **********    WARNING    **************************
#
# Please ensure the P&R netlist name is represented correctly in this tcl file.
# You may need to change the file_name variable to match your actual netlist
# name.
#
###############################################################################

# Set the file_name and  project_name variable
set file_name turboTop.vqm
set project_name turboTop

# Close the project if open
if [is_project_open] {
   project_close
}

# Create a new project
project_new -overwrite -family STRATIXII -part EP2S30F484C3 $project_name

# Make global assignments
set_global_assignment -name TOP_LEVEL_ENTITY $project_name

###############################################################################
# if you are using Verilog P&R netlist, please comment out EDIF assignment
# and uncomment the VERILOG assignment below.

#set_global_assignment -name EDIF_FILE $file_name
set_global_assignment -name  VQM_FILE $file_name
###############################################################################
```

```
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP ON
#set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP OFF
set_global_assignment -name EDA_DESIGN_ENTRY_SYNTHESIS_TOOL -value "Design Compiler FPGA"
set_global_assignment -name EDA_INPUT_VCC_NAME -value VDD -section_id
eda_design_synthesis
set_global_assignment -name EDA_INPUT_GND_NAME -value GND -section_id
eda_design_synthesis
set_global_assignment -name EDA_LMF_FILE -value dc_fpga.lmf -section_id
eda_design_synthesis
set_global_assignment -name VERILOG_LMF_FILE dc_fpga.lmf
set_global_assignment -name FITTER_EFFORT "STANDARD FIT"

# Source in the design timing constraint file
source $project_name\_cons.tcl

# The following runs quartus_map, quartus_fit, quartus_tan, & quartus_asm
load_package flow
execute_flow -compile
project_close
```

After synthesis in the DC FPGA software, the technology-mapped design is written to the current project directory as an Verilog Quartus Mapping netlist file. The project configuration script (<*user-specified file name*>**.tcl**) is used to create and compile a Quartus II project containing your Verilog Quartus Mapping netlist. The example script makes basic project assignments such as assigning the target device as specified in the DC FPGA software. The project configuration script calls the place-and-route constraints script to make your timing constraints. The place-and-route constraints script (<*top module*>_const**.tcl**) forward-annotates the timing constraints that you made in the DC FPGA software, including false path assignments, multi-cycle assignments, timing groups, and related clocks. This integration means that you need to enter these constraints only once, in the DC FPGA software, and they are passed automatically to the Quartus II software.

Place & Route with the Quartus II Software

After you have created your Quartus II project and successfully loaded your Verilog Quartus Mapping netlist into the Quartus II project, you can use the Quartus II software to perform place-and-route. The Synopsys DC FPGA software uses only worst case timing delays and constraints, and does not optimize minimum timing requirements. Altera recommends that you add minimum timing constraints and perform minimum timing analysis in the Quartus II software.

For more information about these advance features, area optimization, and timing closure, refer to the *Quartus II Handbook*.

You can use the Quartus II software to obtain accurate prediction of post-conversion f_{MAX} performance and power consumption characteristics when migrating from a high-density FPGA to a cost-optimized, high-volume structured ASIC such as a HardCopy Stratix device.

The Quartus II software place-and-route algorithms can use register packing, register retiming, automatic logic duplication, and what-you-see-is-what-you-get (WYSIWYG) primitive re-synthesis technologies to increase logic utilization in your device and to deliver superior f_{MAX} performance at extremely high logic utilization.

For more information, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook.*

Formality Software Support

Beginning with version 4.2, the Quartus II software interfaces with the Formality software from Synopsys. Formality software verifies logic equivalency between the RTL and DC FPGA post-synthesis netlist, and between the DC FPGA post-synthesis netlist and the Quartus II post-place-and-route netlist. A synthesized verilog netlist generated by the DC FPGA is required to use with formality flow. Formality supports Stratix II, Stratix and Stratix GX device families.

For more information about how to set the required synthesis settings to generate a valid formal verification netlist and to use the Formality software for equivalence checking, refer to the *Synopsys Formality Support* chapter in volume 3 of the *Quartus II Handbook*.

Conclusion

Large FPGA designs require advanced synthesis of their HDL code. Taking advantage of the Synopsys DC FPGA software and the Quartus II software allows you to develop high-performance designs while occupying as little programmable logic resources as possible. The DC FPGA software and Quartus II software combination is an excellent solution for the high density designs using Altera FPGA devices.

Document Revision History

Table 12–3 shows the revision history of this document.

Table 12–3. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Added revision history to the chapter.	
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 11 was formerly chapter 13 in version 5.0.	
May 2005 v5.0.0	Chapter 13 was formerly chapter 11 in version 4.2.	
Dec. 2004 v1.1	• Chapter 12 was formerly Chapter 13 in version 4.1. • Updated information. • New functionary for Quartus II software version 4.2. • Moved figure 12-3 within the chapter.	
June 2004 v1.0	Initial release.	

13. Analyzing Designs with Quartus II Netlist Viewers

QII51013-6.1.0

Introduction

As FPGA designs grow in size and complexity, the ability to analyze how your synthesis tool interprets your design becomes critical. With today's advanced designs, often several design engineers are involved in coding and synthesizing different design blocks, making it difficult to analyze and debug the design. The Quartus® II RTL Viewer, State Machine Viewer, and Technology Map Viewer provide powerful ways to view your initial and fully mapped synthesis results during the debugging, optimization, or the constraint entry process.

The first section in this chapter, "When to Use Viewers: Analyzing Design Problems", describes examples of using the viewers to analyze your design at various stages of the design cycle. The following sections 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. The next sections describe the following activities:

- How to navigate and filter schematics
- How to probe to and from other windows within the Quartus II software
- How to view a timing path from the Timing Analyzer report

The final section, "Debugging HDL Code with the State Machine Viewer" on page 13–49, provides a detailed example that uses the viewer to analyze a design and quickly resolve a design problem.

When to Use Viewers: Analyzing Design Problems

You can use the netlist viewers to analyze your design to determine how it was interpreted by the Quartus II software. This section provides simple examples of how to use the RTL, State Machine, and Technology Map Viewers to analyze problems encountered in the design process.

For more explanation about how the netlist viewers display your design, refer to the following sections:

- "Quartus II Design Flow with the Netlist Viewers"
- "RTL Viewer Overview"
- "State Machine Viewer Overview"
- "Technology Map Viewer Overview"

To see the user interface for the netlist viewers, refer to "Introduction to the User Interface" on page 13–7.

Using the RTL Viewer is a good way to view your initial synthesis results to determine whether you have created the desired logic, and that the logic and connections have been interpreted correctly by the software. You can use the RTL Viewer and the State Machine Viewer to visually check your design 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, you can use the RTL Viewer to trace through the netlist and ensure that the connections and the logic in your design are as expected. You can also use the State Machine Viewer to view state machine transitions and transition equations. Viewing the design can help you find and analyze the source of design problems. If your design looks correct in the RTL Viewer, you know to focus your analysis on later stages of the design process and investigate potential timing violations or issues in the verification flow itself.

You can use the Technology Map Viewer to look at the results at the end of synthesis and technology mapping by running the viewer after performing Analysis & 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 & Synthesis, then both viewers display the same post-mapping netlist.

In addition, you can use the RTL Viewer or Technology Map Viewer to locate the source of a particular signal, which can help you debug your design. Use the navigation techniques described in this chapter to search easily through the design. You can trace back from a point of interest to find the source of the signal and ensure the connections are as expected.

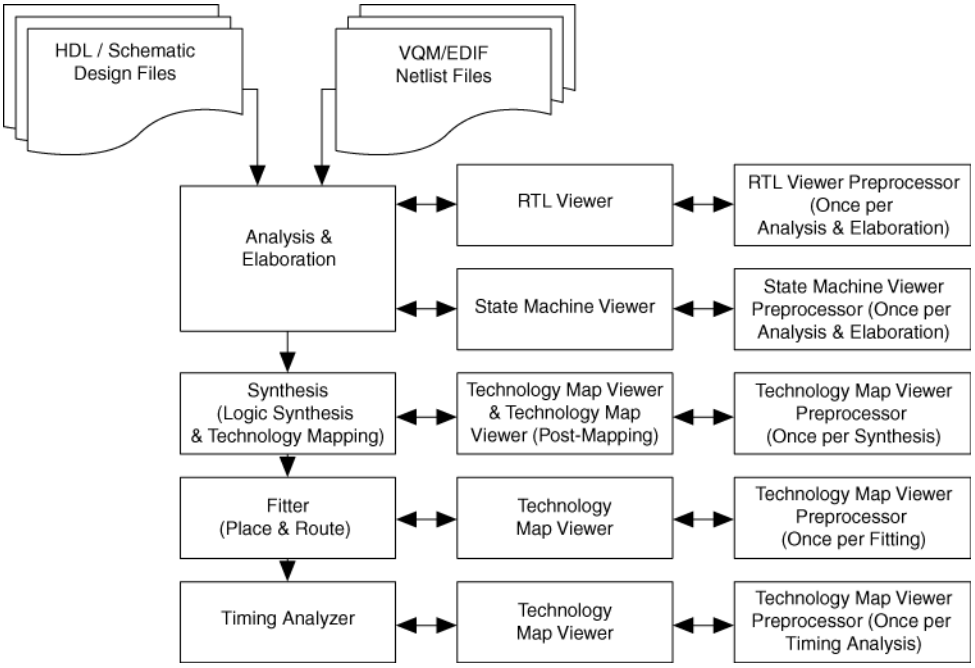
You also can use the Technology Map Viewer to help you locate post-synthesis nodes in your netlist and make assignments when optimizing your design. This functionality is useful, for example, when making a multicycle clock timing assignment between two registers in your design. Start at an I/O port and trace forward or backward through the design and through levels of hierarchy to find nodes that interest you, or locate a specific register by visually inspecting the schematic.

The RTL Viewer, State Machine Viewer, and Technology Map Viewer can be used 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 how you can use the various features of the netlist viewers to increase your productivity when analyzing a design.

Quartus II Design Flow with the Netlist Viewers

The first time you open one of the netlist viewers after compiling the design, a preprocessor stage runs automatically before the viewer opens. If you close the viewer and open it again later without recompiling the design, the viewer opens immediately without performing the preprocessing stage. Figure 13–1 shows how the netlist viewers fit into the basic Quartus II design flow.

Figure 13–1. Quartus II Design Flow Including the RTL Viewer & Technology Map Viewer



Each viewer requires that your design has been compiled with the minimum compilation stage listed below before the viewer can run the preprocessor and open the design.

- To open the RTL Viewer or State Machine Viewer, you must first perform **Analysis & Elaboration**.
- To open the Technology Map Viewer or the Technology Map Viewer (Post-Mapping), you must first perform **Analysis & Synthesis**.

If you open one of the viewers without first compiling the design with the appropriate minimum compilation stage, the viewer does not appear. Instead, the Quartus II software issues an error message instructing you to run the necessary compilation stage and restart the viewer.

Both viewers display the results of the last successful compilation. Therefore, if you make a design change that causes an error during Analysis & Elaboration, you cannot view the netlist for the new design files, but you can still see the results from the last successfully compiled version of the design files. If you receive an error during compilation and you have not yet successfully run the appropriate compilation stage for your project, the viewer cannot be displayed; in this case, the Quartus II software issues an error message when you try to open the viewer.

If the viewer window is open when you start a new compilation, the viewer closes automatically. You must open the viewer again to view the new design netlist after compilation completes successfully.

RTL Viewer Overview

The Quartus II RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your Quartus II integrated synthesis results or your third-party netlist file within the Quartus II software.

You can view results after Analysis & Elaboration when your design uses any supported Quartus II design entry method, including Verilog HDL Design Files (**.v**), VHDL Design Files (**.vhd**), AHDL Text Design Files (**.tdf**), schematic Block Design Files (**.bdf),** or schematic Graphic Design Files (**.gdf)** imported from the MAX+PLUS® II software. You can also view the hierarchy of atom primitives (such as device logic cells and I/O ports) when your design uses a synthesis tool to generate a Verilog Quartus Mapping File (**.vqm**) or Electronic Design Interchange Format (**.edf**) netlist file. Refer to Figure 13–1 for a flow diagram.

The Quartus II RTL Viewer displays a schematic view of the design netlist after analysis and elaboration or netlist extraction is performed by the Quartus II software, but before technology mapping and any synthesis or fitter optimization algorithms occur. This view is not the final design structure because optimizations have not yet occurred. This view most closely represents your original source design. If you synthesized your design using the Quartus II integrated synthesis, this view shows how the Quartus II software interpreted your design files. If you are using a third-party synthesis tool, this view shows the netlist written by your synthesis tool.

When displaying your design, the RTL Viewer optimizes the netlist to maximize readability in the following ways:

- Logic with no fan-out (its outputs are unconnected) and logic with no fan-in (its inputs are unconnected) are removed from the display.
- Default connections such as VCC 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 & Elaboration stage of the Quartus II compilation flow.

To run the viewer, on the Tools menu, point to Netlist Viewers and click **RTL Viewer**, or select **RTL Viewer** from the **Applications** toolbar.

By default, the **Applications** toolbar does not display in the Quartus II user interface. To add the toolbar, on the Tools menu, click **Customize**. On the **Customize** dialog box, click the **Toolbars** tab under **Toolbars**, and turn on **Applications**. Click **Close**.

You can set the RTL Viewer preprocessing to run during a full compilation, which means you can launch the RTL Viewer after Analysis & 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 & 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 the internal registers and look-up tables inside logic cells (LCELLs) and registers in I/O atom primitives. Refer to "Viewing Contents of Atom Primitives in the Technology Map Viewer" on page 13–23 for details.

Where possible, the port names of each hierarchy are maintained throughout synthesis. However, port names may change or be removed from the design. For example, if a port is unconnected or driven by `GND` or `VCC`, it is removed during synthesis. When a port name is changed, the port is assigned a related user logic name in the design, or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, on the Processing menu, point to Start and click **Start Analysis & Synthesis** to synthesize and map the design to the target technology. At this stage, the Technology Map Viewer shows the same post-mapping netlist as does the Technology Map Viewer (Post-Mapping). You also can perform a full compilation, or any process that includes the synthesis stage in the compilation flow.

If you have completed the Fitter stage, the Technology Map Viewer shows the changes made to your netlist by the Fitter, such as physical synthesis optimizations, while the Technology Map Viewer (Post-Mapping) shows the post-mapping netlist. If you have completed the Timing Analysis stage, you can locate timing paths from the Timing Analyzer report in the Technology Map Viewer (refer to "Viewing a Timing Path" on page 13–40 for details). Refer to Figure 13–1 on page 13–3 for a flow diagram.

To run the Technology Map Viewer, on the Tools menu, point to Netlist Viewers and click **Technology Map Viewer**, or select **Technology Map Viewer** from the **Applications** toolbar.

To run the Technology Map Viewer (Post-Mapping), on the Tools menu, point to Netlist Viewers and click **Technology Map Viewer (Post-Mapping)**.

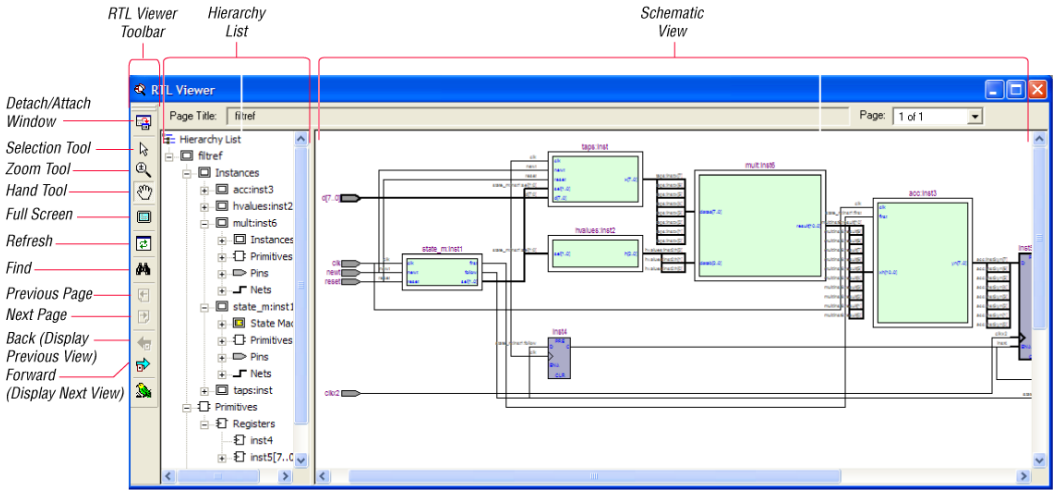
Introduction to the User Interface

The RTL Viewer window and Technology Map Viewer window each consist of two main parts: the schematic view and the hierarchy list. Figure 13–2 shows the RTL Viewer window and indicates these two parts. Both viewers also contain a toolbar that gives you tools to use in the schematic view.

You can have only one RTL Viewer, one Technology Map Viewer, and one State Machine Viewer window open at a time, although each window can show multiple pages. The window for each viewer has characteristics similar to other "child" windows in the Quartus II software; it can be resized and moved, minimized or maximized, tiled or cascaded, and moved in front of or behind other windows.

You can detach the window and move it outside the Quartus II main interface. To detach a window, click the **Detach Window** icon on the toolbar, or, on the Window menu, click **Detach Window**. To attach the detached window back to the Quartus II main interface, click the **Attach Window** icon on the toolbar, or, on the Window menu, click **Attach Window**.

Figure 13–2. RTL Viewer Window & RTL Toolbar



Schematic View

The schematic view is shown on the right side of the RTL Viewer and Technology Map Viewer. It contains a schematic representing the design logic in the netlist. This view is the main screen for viewing your gate-level netlist in the RTL Viewer and your technology-mapped netlist in the Technology Map Viewer.

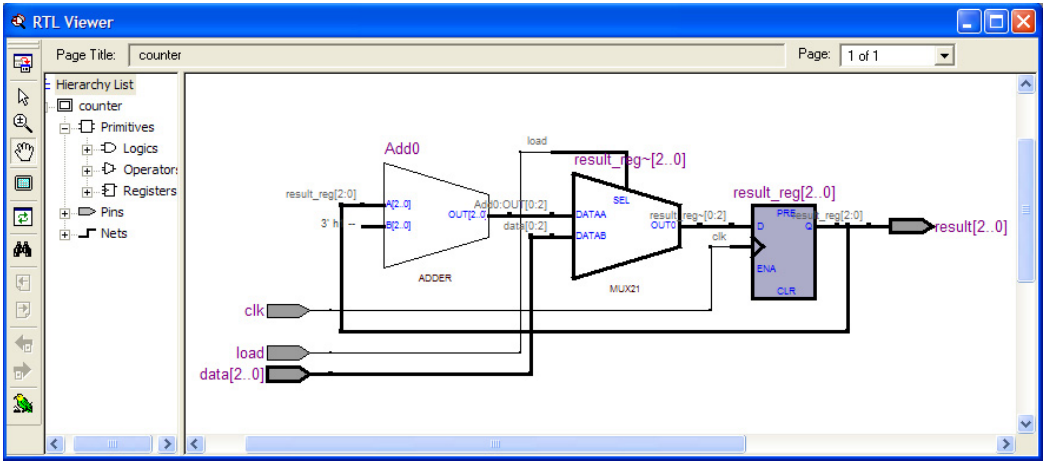
Schematic Symbols

The symbols for nodes in the schematic represent elements of your design netlist. These elements include input and output ports, registers, logic gates, Altera primitives, high-level operators, and hierarchical instances.

Figure 13–3 shows an example of an RTL Viewer schematic for a 3-bit synchronous loadable counter. Example 13–1 shows the Verilog HDL code that produced this schematic. This example includes multiplexers and a group of registers (Table 13–1 on page 13–10) in a bus along with an `ADDER` operator (Table 13–3 on page 13–13) 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, and the COMBOUT port represents the output of the combinational logic in the look-up table (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

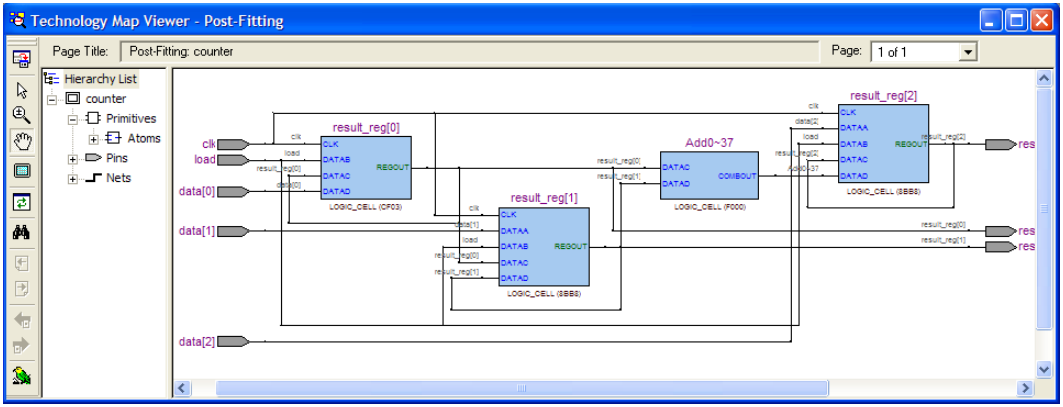


Table 13–1 lists and describes the primitives and basic symbols that you can display in the schematic view of the RTL Viewer and Technology Map Viewer. Table 13–3 on page 13–13 lists and describes the additional higher level operator symbols used in the RTL Viewer schematic view.

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)

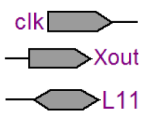
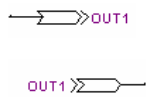
Symbol	Description
I/O Ports 	An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can represent a bus. Only one wire is connected to the bidirectional symbol, representing both the input and the 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 13–25). To go to the page that contains the source or the destination, right-click on the net and choose the page from the menu (refer to "Following Nets Across Schematic Pages" on page 13–27).
Hierarchy Port Connector 	A connector representing a port relationship between two different hierarchies. A connector indicates that a path passes through a port connector in a different level of hierarchy.

Table 13–1. Symbols in the Schematic View (Part 2 of 3)

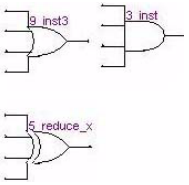
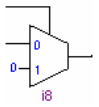
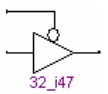
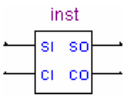
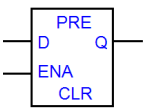
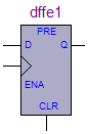
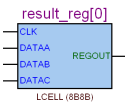
Symbol	Description
OR, AND, XOR Gates 	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output port indicates the port is inverted.
MUX 	A multiplexer (MUX) primitive with a selector port that selects between port 0 and port 1. A MUX with more than two inputs is displayed as an operator (refer to "Operator Symbols in the RTL Viewer Schematic View" on page 13–13).
BUFFER 	A buffer primitive. The figure shows the tri-state buffer, with an inverted output enable port. Other buffers without an enable port include LCELL, SOFT, CARRY, and GLOBAL. The NOT gate and EXP expander buffers use this symbol without an enable port and with an inverted output port.
CARRY_SUM 	A CARRY_SUM buffer primitive with the following ports: ● SI – SUM IN ● SO – SUM OUT ● CI – CARRY IN ● CO – CARRY OUT
LATCH 	A latch primitive with the following ports: ● D – data input ● ENA – enable input ● Q – data output ● PRE – preset ● CLR – clear
DFFE/DFFEA/DFFAES 	A DFFE (data flipflop with enable) primitive, with the same ports as a latch and a clock trigger. The other flipflop primitives are similar: ● DFFEA (data flipflop with enable and asynchronous load) primitive with additional ALOAD asynchronous load and ADATA data signals ● DFFEAS (data flipflop with enable and both synchronous and asynchronous load), which has ASDATA as the secondary data port
Atom Primitive 	Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive type, and its name. The blue shading indicates an atom primitive in the Technology Map Viewer that allows you to view the internal details of the primitive. Refer to "Viewing Contents of Atom Primitives in the Technology Map Viewer" on page 13–23 for details.

Table 13–1. Symbols in the Schematic View (Part 3 of 3)

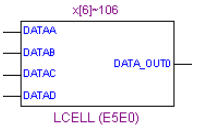
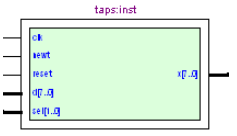
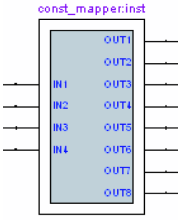
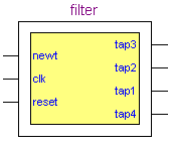
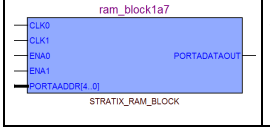
Symbol	Description
Other Primitive 	Any primitive that does not fall into the categories above. Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive or operator type, and its name. The figure shows an LCELL WYSIWYG primitive, with DATAA to DATAD and COMBOUT port connections. This type of LCELL primitive would be 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 was a VQM or EDIF netlist.
Instance 	An instance in the design that does not correspond to a primitive or operator (generally a user-defined hierarchy block), indicated by the double outline and green shading. The symbol displays the instance name. To open the schematic for the lower level hierarchy, right-click and choose the appropriate command (refer to "Traversing & Viewing the Design Hierarchy" on page 13–21).
Encrypted Instance 	A user-defined encrypted instance in the design, indicated by the double outline and gray shading. The symbol displays the instance name. You cannot open the schematic for the lower level hierarchy, because the source design is encrypted.
State Machine Instance 	A finite state machine instance in the design, indicated by the double outline and yellow shading. Double-clicking this instance opens the State Machine Viewer. Refer to "State Machine Viewer" on page 13–18 for more details.
RAM 	A synchronous memory instance with registered inputs and optionally registered outputs, indicated by purple shading. The symbol shows the device family and the type of TriMatrix™ memory block. This figure shows a true dual-port memory block in a Stratix M-RAM block.

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

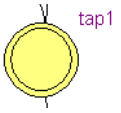
Symbol	Description
State Node 	The node representing a state in a finite state machine. State transitions are indicated with arcs between state nodes. The double circle border indicates the state connects to logic outside the state machine, while a single circle border indicates the state node does not feed outside logic.

Table 13–3 lists and describes the additional higher level operator symbols used in the RTL Viewer schematic view.

Table 13–3. Operator Symbols in the RTL Viewer Schematic View (Part 1 of 2)

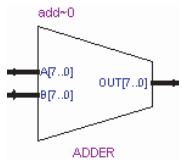
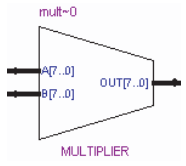
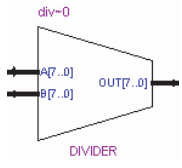
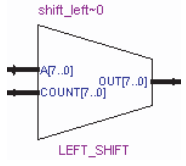
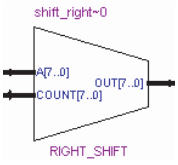
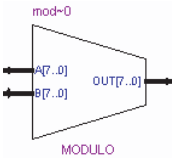
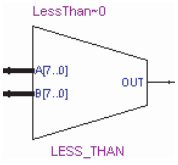
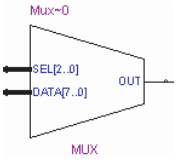
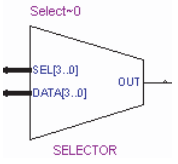
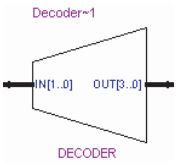
Symbol	Description
	An adder operator: `OUT = A + B`
	A multiplier operator: `OUT = A × B`
	A divider operator: `OUT = A / B`
	A left shift operator: `OUT = (A << COUNT)`

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 viewer toolbar (this tool is enabled by default). Click on an item in the schematic view to highlight it in red.

Select multiple items by pressing the Shift or Ctrl key while selecting with your mouse. You also can select all nodes in a region by selecting a rectangular box area with your mouse cursor when the **Selection Tool** is enabled. To select nodes in a box, move your mouse to one corner of the area you want to select, click the mouse button, and drag the mouse to the opposite corner of the box, then release the mouse button. By default, creating a box like this highlights and selects all nodes in the selected area (instances, primitives, and pins), but not the nets. The **Viewer Options** dialog box provides an option to select nets. To include nets, right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Select entire net when segment is selected** option.

Items selected in the schematic view are automatically selected in the hierarchy list (refer to the "Hierarchy List" on page 13–16). The list expands automatically if required to show the selected entry. However, the list does not collapse automatically when entries are not being used or are deselected.

When you select a hierarchy box, node, or port in the schematic view, the item is highlighted in red but none of the connecting nets are highlighted. When you select a net (wire or bus) in the schematic view, all connected nets are highlighted in red. The selected nets are highlighted across all hierarchy levels and pages. Net selection can be useful when navigating a netlist because you see the net highlighted when you traverse between hierarchy levels or pages.

In some cases, when you select a net that connects to nets in other levels of the hierarchy, these connected nets also are highlighted in the current hierarchy. If you prefer that these nets not be highlighted, use the **Viewer Options** dialog box option to highlight a net only if the net is in the current hierarchy. Right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Limit selections to current hierarchy** option.

Moving & Panning in the Schematic View

When the schematic view page is larger than the portion currently displayed, you can use the scroll bars at the bottom and right side of the schematic view to see other areas of the page.

You can also use the Hand Tool to "grab" the schematic page and drag it in any direction. Enable the Hand Tool with the toolbar button. Click and drag to move around the schematic view without using the scroll bars.

In addition to the scroll bars and Hand Tool, you can use the middle-mouse/wheel button to move and pan in the schematic view. Click the middle-mouse/wheel button once to enable the feature. Move the mouse or scroll the wheel to move around the schematic view. Click the middle-mouse/wheel button again to turn the feature off.

Schematic Debugging & Tracing Using the Bird's Eye View

Viewing the entire schematic may be useful when debugging and tracing through a large netlist. The Quartus II software allows you to view the schematic from a bird's eye perspective. The bird's eye view is displayed in a separate window that is linked directly to the netlist viewers. This feature is available in the RTL, Technology Map, and Technology Map (Post-Mapping) viewers.

The bird's eye view is a rectangular window that shows the current area of interest. Select the desired area by moving the indicator or using the right-mouse button to form a rectangular box around the desired area. To open the bird's eye view, on the View menu, click **Bird's Eye View**, or click on the Bird's Eye View icon in the Viewer toolbar (Figure 13–5).

Figure 13–5. Bird's Eye View Icon

Hierarchy List

The hierarchy list is displayed on the left side of the viewer window. The hierarchy list displays the entire netlist in a tree format based on the hierarchical levels of the design. Within each level, similar elements are grouped into sub-categories. Using the hierarchy list, you can traverse through the design hierarchy to view the logic schematic for each level. You also can select an element in the hierarchy list to be highlighted in the schematic view.

Nodes inside atom primitives are not listed in the hierarchy list.

For each module in the design hierarchy, the hierarchy list displays the applicable elements listed in Table 13–4. Click the + icon to expand an element.

Table 13–4. Hierarchy List Elements

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 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 can not 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.

Selecting an Item in the Hierarchy List

When you click any item in the hierarchy list, the viewer performs the following actions:

- Searches for the item in the currently viewed pages, and displays the page containing the selected item in the schematic view if it is not currently displayed. (If you are currently viewing a filtered netlist, for example, the relevant page within the filtered netlist is displayed.)
- If the selected item is not found in the currently viewed pages, the entire design netlist is searched, and the item is displayed in a default view.
- Highlights the selected item in red in the schematic view.

When you double-click an instance in the hierarchy list, the viewer displays the underlying implementation of the instance.

You can select multiple items by pressing the Shift or Ctrl key while selecting with your mouse. When you right-click an item in the hierarchy list, you can navigate in the schematic view using the **Filter** and **Locate** commands. Refer to "Filtering in the Schematic View" on page 13–29 and "Probing to Source Design File & Other Quartus II Windows" on page 13–37 for more information.

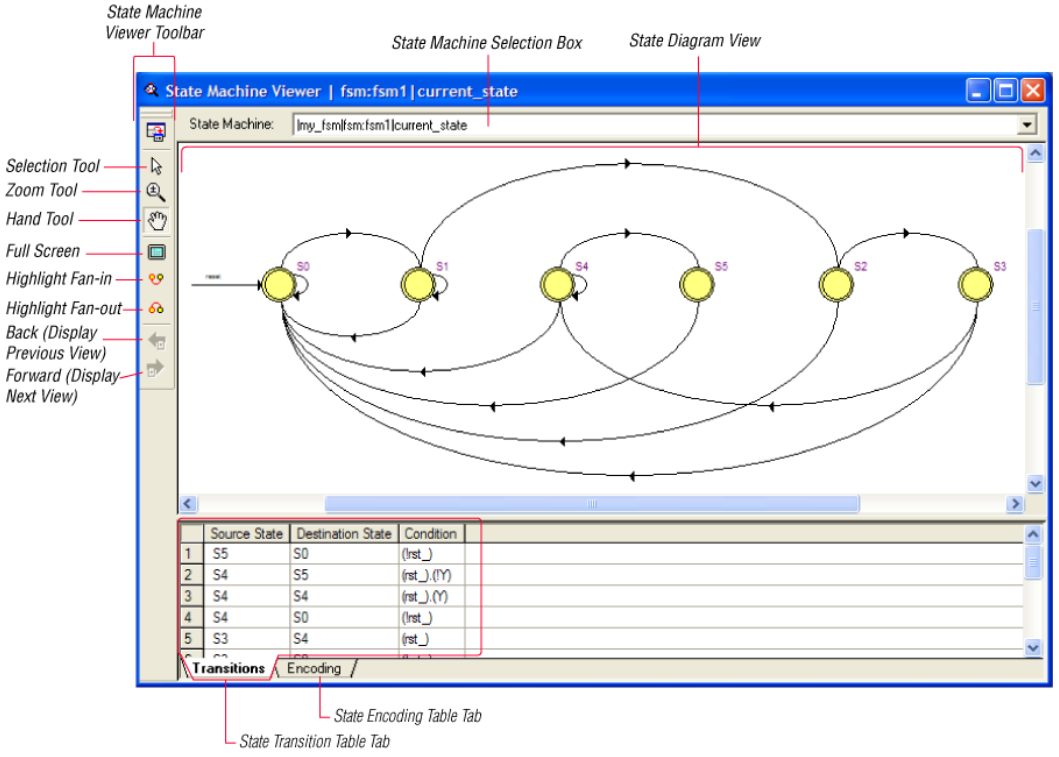
State Machine Viewer

The State Machine Viewer displays a graphical representation of the state machines in your design. You can open the State Machine Viewer in any of the following ways:

- On the Tools menu, point to Netlist Viewers, and click **State Machine Viewer**
- Double-click on a state machine instance in the RTL Viewer
- Right-click on a state machine instance in the RTL Viewer, and click **Hierarchy Down**
- Select a state machine instance in the RTL Viewer, and on the Project menu, point to Hierarchy and click **Down**

Figure 13–6 shows an example of the State Machine Viewer for a simple state machine. The State Machine toolbar on the left side of the viewer provides tools you can use in the state diagram view.

Figure 13–6. State Machine in the State Machine Viewer



State Diagram View

The state diagram view is shown at the top of the State Machine Viewer window. It contains a diagram of the states and state transitions.

The nodes that represent each state are arranged horizontally in the state diagram view with the initial state (the state node that receives the reset signal) in the left-most position. Nodes that connect to logic outside of the state machine instance are represented by a double circle. The state transition is represented by an arc with an arrow pointing in the direction of the transition.

When you select a node in the state diagram view, if you turn on the **Highlight Fan-in** or **Highlight Fan-out** command from the View menu or the State Machine Viewer toolbar, the respective fan-in or fan-out transitions from the node are highlighted in red.

An encrypted block with a state machine displays encoding information in the state encoding table, but does not display a state transition diagram or table.

State Transition Table

The state transition table on the **Transitions** tab at the bottom of the State Machine Viewer window displays the condition equation for each state transition. Each transition (each arc in the state diagram view) is represented by a row in the table. The table has the following three columns:

- **Source State**—the name of the source state for the transition
- **Destination State**—the name of the destination state for the transition
- **Condition**—the condition equation that causes the transition from source state to destination state

To see all of the transitions to and from each state name, click the appropriate column heading to sort on that column.

The text in each column is left-aligned by default; to change the alignment and more easily see the relevant part of the text, right-click in the column and click **Align Right**. To change back to left alignment, click **Align Left**.

You can click in any cell in the table to select it. To select all cells, right-click in the cell and click **Select All**; or, on the Edit menu, click **Select All**. To copy selected cells to the clipboard, right-click the cells and click **Copy Table**; or, on the Edit menu, point to Copy and click **Copy Table**. You can paste the table into any text editor as tab-separated columns.

State Encoding Table

The state encoding table on the **Encoding** tab at the bottom of the State Machine Viewer window displays the encoding information for each state transition.

To view state encoding information in the State Machine Viewer, you must have synthesized your design using **Start Analysis & Synthesis**. If you have only elaborated your design using **Start Analysis & Elaboration**, the encoding information is not displayed.

Selecting an Item in the State Machine Viewer

You can select and highlight each state node and transition in the State Machine Viewer. To select a state transition, click the arc that represents the transition.

When you select a state node, transition arc, or both in the state diagram view, the matching state node and equation conditions in the state transition table are highlighted. Conversely, when you select a state node, equation condition, or both in the state transition table, the corresponding state node and transition arc are highlighted in the state diagram view.

Switching Between State Machines

A design may contain multiple state machines. To choose which state machine to view, use the **State Machine** selection box located at the top of the State Machine Viewer. Click in the drop-down box and select the desired state machine.

Navigating the Schematic View

The previous sections provided an overview of the user interface for each netlist viewer, and how to select an item in each viewer. This section describes methods to navigate through the pages and hierarchy levels in the schematic view of the RTL Viewer and Technology Map Viewer.

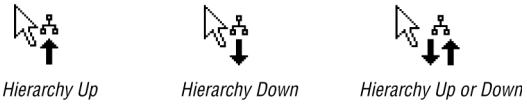
Traversing & Viewing the Design Hierarchy

You can open different hierarchy levels in the schematic view using the hierarchy list (refer to "Hierarchy List" on page 13–16), or the **Hierarchy Up** and **Hierarchy Down** commands in the schematic view.

Use the **Hierarchy Down** command to go down into, or expand an instance's hierarchy, and open a lower level schematic showing the internal logic of the instance. Use the **Hierarchy Up** command to go up in hierarchy, or collapse a lower level hierarchy, and open the parent higher level hierarchy. When the **Selection Tool** is selected, the appropriate option is available when your mouse pointer is located over an area of the schematic view that has a corresponding lower or higher level hierarchy.

The mouse pointer changes as it moves over different areas of the schematic to indicate whether you can move up, down, or both up and down in the hierarchy (Figure 13–7). 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 the lower level hierarchies of the current design hierarchy, right-click in the schematic and click **Flatten Netlist**. To flatten the entire design, choose **Flatten Netlist** from the top-level schematic of the design.

Viewing the Contents of a Design Hierarchy within the Current Schematic

You can use the **Display Content** and **Hide Content** commands to show or hide a lower hierarchy level for a specific instance within the schematic for the current hierarchy level.

To display the lower hierarchy netlist of an instance on the same schematic as the remaining logic in the currently viewed netlist, right-click the selected instance and click **Display Content**.

To hide all of the lower hierarchy logic of a hierarchy box into a closed instance, right-click the selected instance and click **Hide Content**.

Customizing the Schematic Display

You can customize the schematic display for better viewing and to speed up your debugging process. The options that control the schematic display are available in the **Customize View** tab of the RTL/Technology Map Viewer Options dialog box. To open the dialog box, right-click in the schematic and click **Viewer Options**. You can turn on the options to remove fan-out free nodes, simplify logic, and group or ungroup related nodes.

When the settings are changed, the list of previously viewed pages is cleared. The settings are revision-specific, so different revisions could have different settings. An hourglass figure shows that processing is in progress.

To remove fan-out free registers in your schematic display, turn on **Remove registers without fan-out**. By default, this option is turned on.

To remove all single-input nodes and merge a chain of equivalent combinational gates that have direct connections (without inversion in between) into a single multiple-inputs gate, turn on **Show simplified logic.** By default, this option is turned on.

To group all related nodes into a single node, turn on **Group all related nodes**. This option is turned on by default. You can manually group or ungroup any nodes by right-clicking the selected nodes in the schematic and selecting **Group Related Nodes** to group, or **Ungroup Selected Nodes** to ungroup.

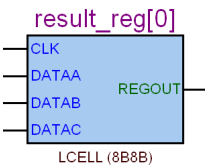
Viewing Contents of Atom Primitives in the Technology Map Viewer

In the Technology Map Viewer, you can view the contents of certain device atom primitives to see their underlying implementation details. For logic cell (`LCELL`) atoms in the Stratix and Cyclone® series of devices and in MAX® II devices, you can view the LUTs, registers, and logic gates. For I/O atoms in the Stratix and Cyclone series of devices, and HardCopy® II devices, you can view the registers and logic gates.

In addition, you can view the implementation of RAM and DSP blocks in certain devices. You can view the implementation of RAM blocks in the Stratix and Cyclone series of devices. You can view the implementation of DSP blocks only in the Stratix series of devices.

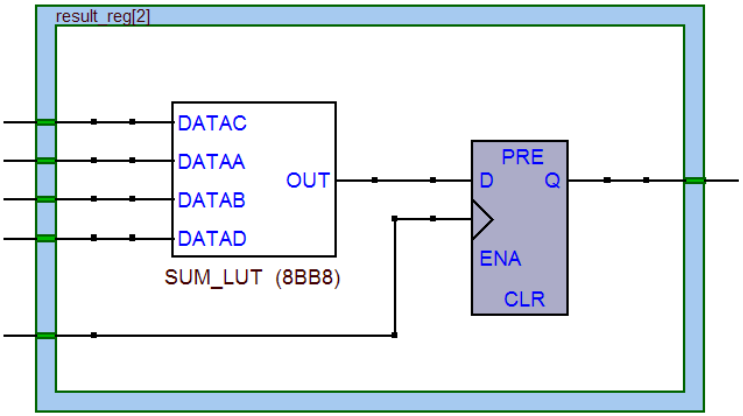
If you can view the contents of an atom instance, it is blue in the schematic view (Figure 13–8).

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 desired atom instances. Right-click a selected instance and click **Display Content**. You also can double-click on the desired 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(s), and click **Hide Content**.

In the schematic view, the internal details within an atom instance can not be selected as individual nodes. Any mouse action on any of the internal details is treated as a mouse action on the atom instance.

Zooming & Magnification

You can control the magnification of your schematic with the View menu, the **Zoom Tool** in the toolbar, or the Ctrl key and mouse wheel button, as described in this section.

The **Fit in Window**, **Fit Selection in Window**, **Zoom In**, **Zoom Out**, and **Zoom** commands are available from the View menu, by right-clicking in the schematic view and selecting **Zoom,** or from the **Zoom** toolbar. To enable the zoom toolbar, on the Tools menu, click **Customize**. Click the **Toolbars** tab and click **Zoom** to enable the toolbar.

By default, the viewer displays most pages sized to fit in the window. If the schematic page is very large, the schematic is displayed at the minimum zoom level, and the view is centered on the first node. Select **Zoom In** to view the image at a larger size, and select **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 within the window. Use the **Selection Tool** to select one or more nodes (instances, primitives, pins, and nets), then select **Fit Selection in Window** to enlarge the area covered by the selection. This feature is helpful when you want to see a particular element in a large schematic. After you select a node, you can easily zoom in to view the particular node.

You also can use the **Zoom Tool** on the viewer toolbar to control magnification in the schematic view. When you select the **Zoom Tool** in the toolbar, clicking on the schematic zooms in and centers the view on the location you clicked. Right-click on the schematic (or press the Shift key or the Ctrl key and click) to zoom out and center the view on the location you clicked. When you select the **Zoom Tool**, you also can 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. To change the minimum and the maximum zoom level, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers**, and set the desired minimum and maximum zoom level.

By default, the viewers maintain the zoom level when filtering on the schematic (refer to "Filtering in the Schematic View" on page 13–29). To change the behavior so that the zoom level is always reset to "Fit in Window," on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers**, and turn off **Maintain zoom level**.

Partitioning the Schematic into Pages

For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view. To control how much of the design is visible on each page, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers**, and set the desired options under **Display Settings**.

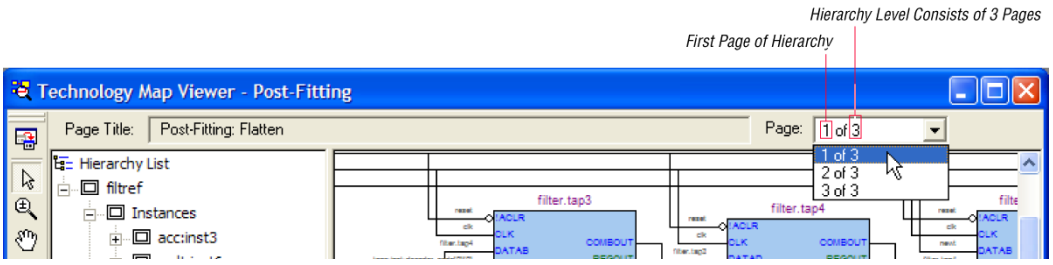
The **Nodes per page** option specifies the number of nodes per partitioned page. The default value is 50 nodes; the range is 1 to 1,000 nodes. The **Ports per page** option specifies the number of ports (or pins) per partitioned page. The default value is 1,000 ports (or pins); the range is 1

to 2,000 ports (or pins). The viewers partition your design into a new page if either the node number or the port number exceeds the limit you have specified. You may occasionally see the number of ports exceed the limit, depending on the configuration of nodes on the page.

If the **Display boundary around hierarchy levels** option is turned on, and the total number of nodes or ports within the hierarchy exceeds the value of **Nodes per page** or **Ports per page**, the boundary is displayed as a hierarchy port connector (refer to Table 13–1 on page 13–10). For more information about the **Display boundary around hierarchy levels** option, refer to "Filtering Across Hierarchies" on page 13–33.

When a hierarchy level is partitioned into multiple pages, the title bar for the schematic window indicates which page is displayed and how many total pages exist for this level of hierarchy (shown in the format: Page <*current page number*> of <*total number of pages*>) as shown in Figure 13–10.

Figure 13–10. RTL Viewer Title Bars Indicating Page Number Information



When you change the number of nodes or ports per page, the change applies only to new pages that are shown or opened in the viewer. To refresh the current page so that it displays the changed number of nodes or ports, click the **Refresh** button in the toolbar.

Moving Between Schematic Pages

To move to another schematic page, on the View menu, click **Previous Page** or **Next Page**, or click the **Previous Page** icon or the **Next Page** icon in the viewer toolbar.

To go to a particular page of the schematic, on the Edit menu, click **Go To**, or right-click in the schematic view, and click **Go To**. In the **Page** list, select the desired page number. You can also go to a particular page by selecting the desired page number from the drop down list on the top right of the viewer window.

Moving Back & Forward Through Schematic Pages

To return to the previous view after changing the page view, click **Back** on the View menu, or click the **Back** icon on the viewer toolbar. To go to the next view, click **Forward** on the View menu, or click the **Forward** icon on the viewer toolbar.

You can go **Forward** only if you have not made any changes to the view since going **Back**. Use **Back** and **Forward** to switch between page views. These commands do not undo an action such as selecting a node.

Following Nets Across Schematic Pages

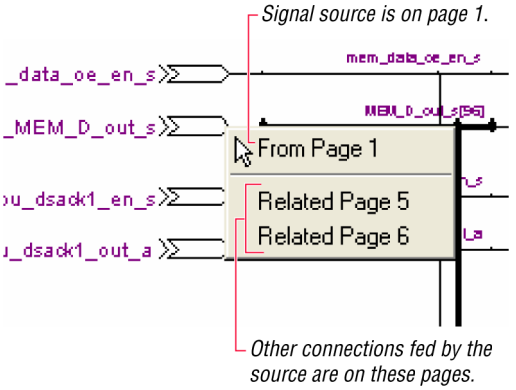
Input and output connectors indicate nodes that connect across pages of the same hierarchy. Right-click on a connector to display a menu of commands that trace the net through the pages of the hierarchy.

After you right-click to follow a connector port, the viewer opens a new page, which centers the view on the particular source or destination net using the same zoom factor used by the previous page. To trace a specific net to the new page of the hierarchy, Altera recommends that you first select the desired net, which highlights it in red, before you right-click to traverse pages.

Input Connectors

Figure 13–11 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.

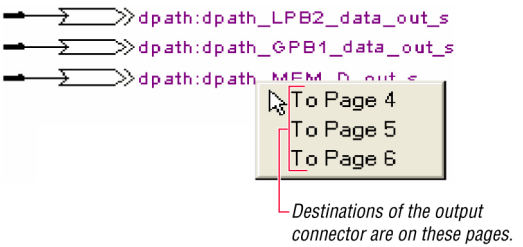
Figure 13–11. Input Connector Right Button Pop-Up Menu



Output Connectors

Figure 13–12 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 13–12. Output Connector Right Button Pop-Up Menu



Go to Net Driver

To locate the source of a particular net in the schematic view, select the net to highlight it, right-click the selected net, point to Go to Net Driver, and click **Current page**, **Current hierarchy**, or **Across hierarchies**. Refer to Table 13–5 for details.

Table 13–5. Go to Net Driver Commands

Command	Action
Current page	Locates the source or driver on the current page of the schematic only.
Current hierarchy	Locates the source within the current level of hierarchy, even if the source is located on another page of the netlist schematic.
Across hierarchies	Locates the source across hierarchies until the software reaches the source at the top hierarchy level.

The schematic view opens the correct page of the schematic if needed, and adjusts the centering of the page so that you can see the net source. The schematic shows the default page for the net driver. The view is an unfiltered view, so no filtering results are kept.

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only a logic path that interests you.

Filter your netlist by selecting hierarchy boxes, nodes, ports of a node, net, or states in a state machine that are part of the path you want to see. The following filter commands are available:

- **Sources**—Displays the sources of the selection
- **Destinations**—Displays the destinations of the selection
- **Sources & Destinations**—Displays both the sources and destinations of the selection
- **Selected Nodes and Nets**—Displays only the selected nodes and nets with the connections between them
- **Between Selected Nodes**—Displays nodes and connections in the path between the selected nodes
- **Bus Index**—Displays the sources or destinations for one or more indices of an output or input bus port

Select a hierarchy box, node, port, net, or state node, right-click in the window, point to Filter and click the appropriate filter command. The viewer generates a new page showing the netlist that remains after filtering.

When filtering in a state diagram in the State Machine Viewer, sources and destinations refer to the previous and next transition states or paths between transition states in the state diagram. The transition table and encoding table also reflect the filtering.

You can go back to the netlist page before it was filtered using the **Back** command, described in "Moving Back & Forward Through Schematic Pages" on page 13–27.

When viewing a filtered netlist, clicking an item in the hierarchy list causes the schematic view to display an unfiltered view of the appropriate hierarchy level. You cannot use the hierarchy list to select items or navigate in a filtered netlist.

Filter Sources Command

To filter out all but the source of the selected item, right click the item, point to Filter and click **Sources**. The selected object type determines what is displayed, as outlined in Table 13–6, and shown in Figure 13–13 on page 13–31.

Table 13–6. 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 13–13 on page 13–31.
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 13–7, and shown in Figure 13–13 on page 13–31, right-click the node or port, point to Filter, and click **Destinations**.

Table 13–7. Selected Objects Determine Filter Destinations Display (Part 1 of 2)

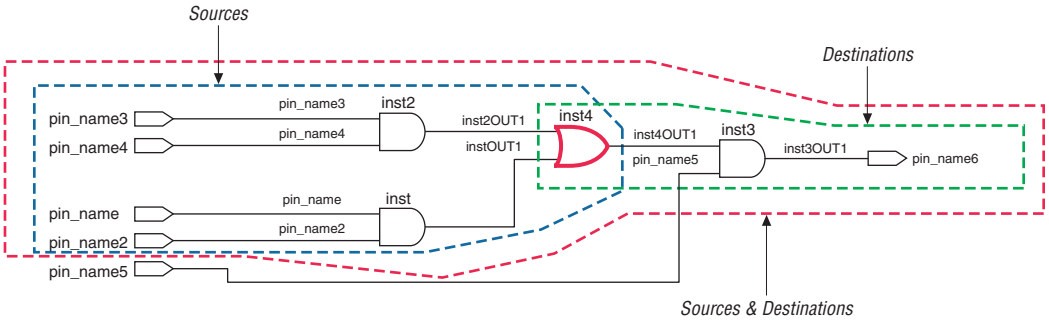
Selected Object	Result Shown in Filtered Page
Node or hierarchy box	Shows all the destinations of the node's output ports. For an example, refer to Figure 13–13 on page 13–31.
Net	Shows the destinations fed by the net.

Table 13–7. Selected Objects Determine Filter Destinations Display (Part 2 of 2)	
Selected Object	**Result Shown in Filtered Page**
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 & Destinations Command

The **Sources & Destinations** command is a combination of the **Sources** and **Destinations** filtering commands, in which the filtered page shows both the sources and the destinations of the selected item. To select this option, right-click on the desired object, point to Filter, and click **Sources & Destinations**. Refer to the example in Figure 13–13.

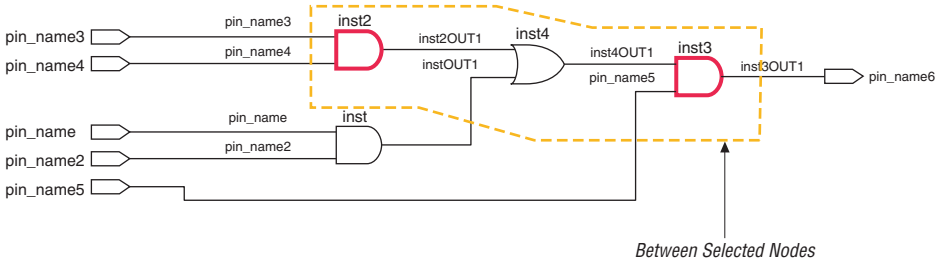
Figure 13–13. Sources, Destinations, and Sources & Destinations Filtering for inst4



Filter Between Selected Nodes Command

To show the nodes in the path between two or more selected nodes or hierarchy boxes, right-click, point to Filter, and click **Between Selected Nodes**. For this option, selecting a port of a node is the same as selecting the node. For an example, refer to Figure 13–14.

Figure 13–14. Between Selected Nodes Filtering Between inst2 & inst3



Filter Selected Nodes & Nets Command

To create a filtered page that shows only the selected nodes, nets, or both, and, if applicable, the connections between the selected nodes, nets, or both, right-click, point to Filter, and click **Selected Nodes & Nets**. Figure 13–15 shows a schematic with several nodes selected.

Figure 13–15. Using Selected Nodes & Nets to Select Nodes

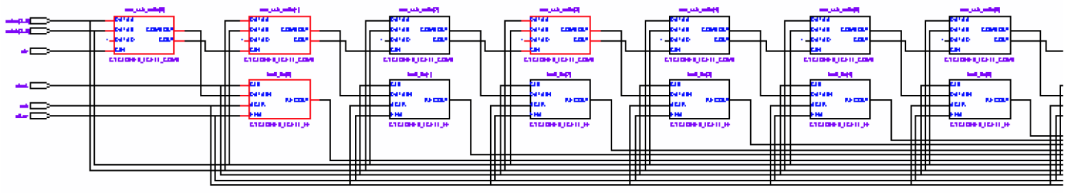
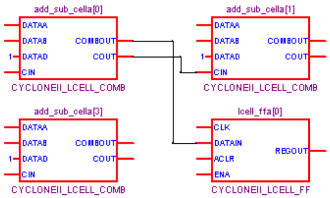


Figure 13–16 shows the schematic after filtering has been performed. If you select a net, the filtered page shows the immediate sources and destinations of the selected net.

Figure 13–16. Selected Nodes & Nets Filtering on Figure 13–15 Schematic



Filter Bus Index Command

To show the path related to a specific index of a bus input or output port in the RTL Viewer, right-click the port, point to Filter, and click **Bus Index**. The **Select Bus Index** dialog box allows you to select the indices of interest.

Filter Command Processing

The options to control filtering are available in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic, and click **Viewer Options** to open the dialog box.

For all the filtering commands, the viewer stops tracing through the netlist to obtain the filtered netlist when it reaches one of the following objects:

- A pin
- A specified number of filtering levels, counting from the selected node or port; the default value is 3

  Specify the **Number of filtering** levels in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. The default value is 3 to ensure optimal processing time when performing filtering, but you can specify a value from 1 to 100.

- A register (optional; turned on by default)

  Turn the **Stop filtering at register** option on or off in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and click **Viewer Options** to open the dialog box.

By default, the filtered schematic shows all possible connection between the nodes shown in the schematic. To remove the connections that are not directly part of the path that was traced to generate a filtered netlist, turn off the **Shows all connections between nodes** option in the **Tracing** section of the **RTL/Technology Map Viewer Option**s dialog box.

Filtering Across Hierarchies

The filtering commands display nodes in all hierarchies by default. When the filtered path passes through levels of hierarchy on the same schematic page, green hierarchy boxes group the logic and show the hierarchy

boundaries. A green rectangular symbol appears on the border that represents the port relationship between two different hierarchies (Figure 13–17 and Figure 13–18).

The **RTL/Technology Map Viewer Options** dialog box provides an option to control filtering if you prefer to filter only within the current hierarchy. Right-click in the schematic, and click **Viewer Options**. In the **Tracing** section, turn off the **Filter across hierarchy** option.

To disable the box hierarchy display, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn off **Display boundary around hierarchy levels**.

Netlists of the same hierarchy that are displayed over more than one page are not grouped with a box. Filtering and expanding on a blue atom primitive does not trace the underlying netlist even when **Filter across hierarchy** is enabled.

Figures 13–17 and 13–18 show examples of filtering across hierarchical boundaries. Figure 13–17 shows an example after the **Sources** filter has been applied to an input port of the `taps` instance, where the input port of the lower level hierarchical block connects directly to an input pin of the design. The name of the instance is indicated within the green border and appears as a tooltip when you move your mouse pointer over the instance.

Figure 13–17. Filtering Across Hierarchical Boundaries, Small Example

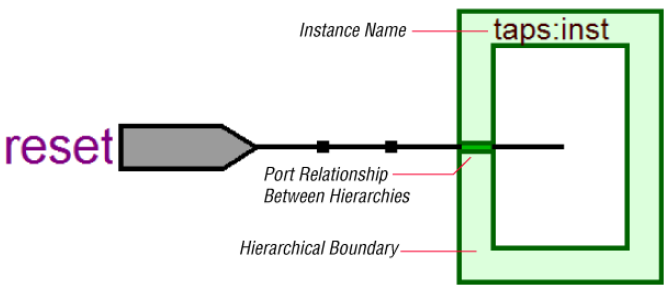
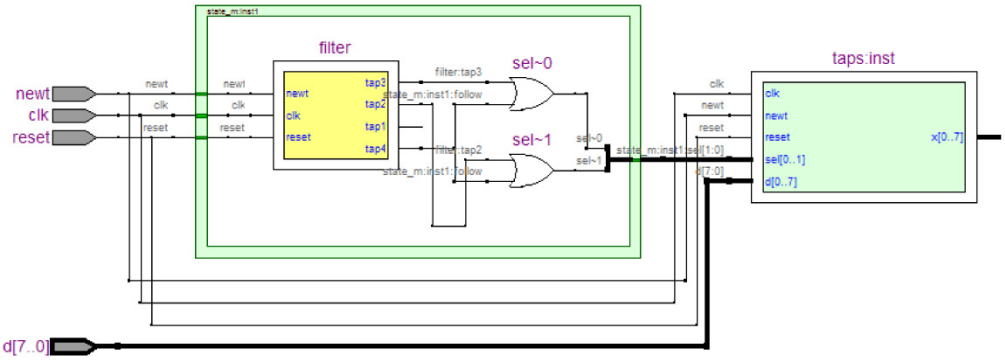


Figure 13–18 shows a larger example after the **Sources** filter has been applied to an input port of an instance, in which the source comes from input pins that are fed through another level of hierarchy.

Figure 13–18. Filtering Across Hierarchical Boundaries, Large Example



Sources command applied to an input port of an instance in which the source comes from input pins that are fed through another level of hierarchy.

Expanding a Filtered Netlist

After a netlist is filtered, some ports may have no 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 the **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 100 levels). You also can set the **Stop expanding at register** option (which is turned on by default) to specify whether netlist expansion should stop when a register is reached.

You can select multiple nodes to expand when using the **Expand** command. If you select ports that are located on multiple schematic pages, only the ports on the currently viewed page appear in the expanded schematic.

In the State Machine Viewer, the **Expand** command has the following three options:

- **Sources**—Displays the states that feed the selected states (previous transition states)
- **Destinations**—Displays the states that are fed by the selected states (next transition states)
- **Sources & Destinations**—Displays both the previous and next transition states

The state transition table and state encoding table also reflect the changes to the filtering.

The expansion feature works across hierarchical boundaries if the filtered page containing the port to be expanded was generated with the **Filter across hierarchy** option turned on (refer to "Filtering in the Schematic View" on page 13–29 for details on this option). When viewing timing paths in the Technology Map Viewer, the **Expand** command always works across hierarchical boundaries because filtering across hierarchy is always turned on for these schematics (refer to "Viewing a Timing Path" on page 13–40 for details on these schematics).

Reducing a Filtered Netlist

In some cases, removing logic from a filtered schematic or state diagram makes the schematic view easier to read or minimizes distracting logic that you do not need to view in the schematic.

To reduce elements in the filtered schematic or state diagram view, right-click the node or nodes you want to remove and click **Reduce**.

Probing to Source Design File & Other Quartus II Windows

The RTL, Technology Map, and State Machine Viewers let you cross-probe from the viewer to the source design file and to various other windows within the Quartus II software. You can select one or more hierarchy boxes, nodes, nets, state nodes, or state transition arcs that interest you in the viewer and locate the corresponding items in another applicable Quartus II software window. You then can view and make changes or assignments in the appropriate editor or floorplan.

To locate an item from the viewer in another window, right-click the items of interest in the schematic or state diagram view, point to Locate, and click the appropriate command. The following commands are available:

- **Locate in Assignment Editor**
- **Locate in Pin Planner**
- **Locate in Timing Closure Floorplan**
- **Locate in Chip Planner**
- **Locate in Resource Property Editor**
- **Locate in RTL Viewer**
- **Locate in Technology Map Viewer**
- **Locate in Design File**

The options available for locating depend on the type of node and whether it exists after placement and routing. If a command is enabled in the menu, then it is available for the selected node. You can use the **Locate in Assignment Editor** command for all nodes, but assignments may be ignored during placement and routing if they are applied to nodes that do not exist after synthesis.

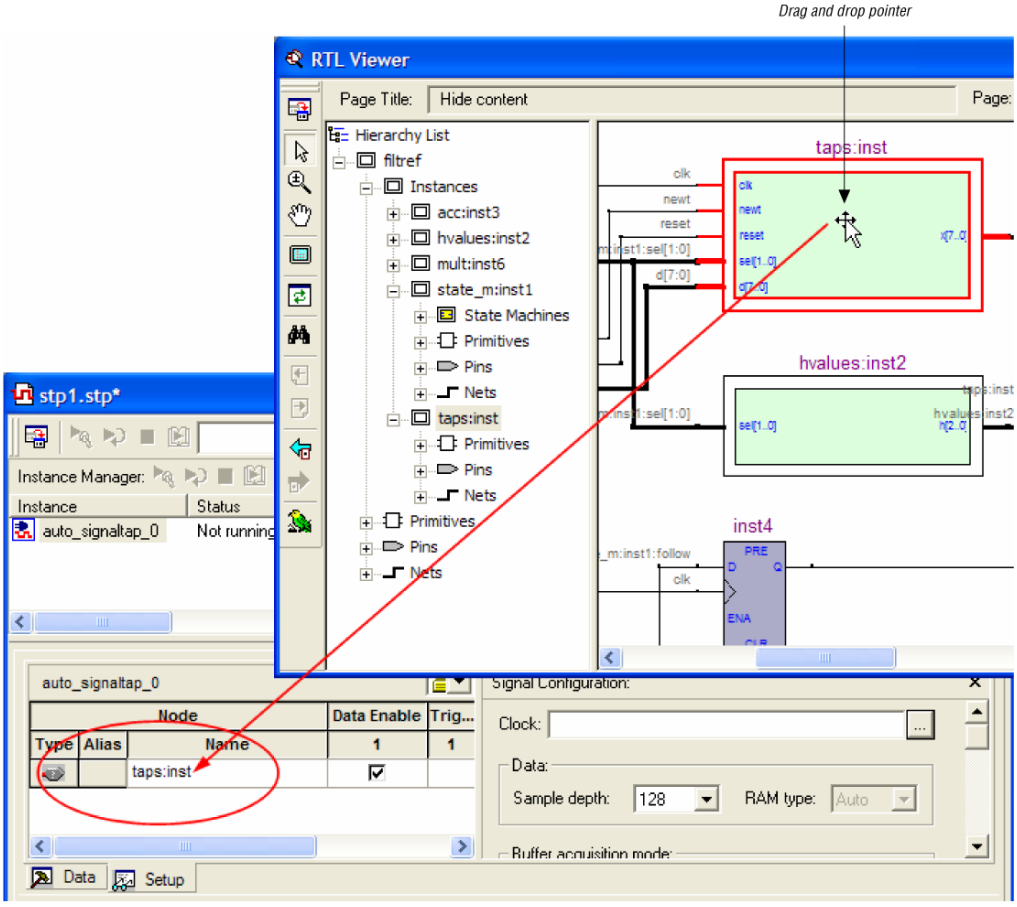
The viewer automatically opens another window for the appropriate editor or floorplan, and highlights the selected node or net in the newly opened window. You can switch back to the viewer by selecting it in the Window menu or by closing, minimizing, or moving the new window.

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, and Waveform Editor windows within the Quartus II software. Whenever you see the drag and drop pointer on the selected node in the netlist viewers, it means that the node can be dragged to other child windows within the Quartus II software.

Figure 13–19 shows the drag and drop pointer and an example of dragging a node from the RTL Viewer to the SignalTap II Logic Analyzer.

Figure 13–19. Dragging a Node to the SignalTap II Logic Analyzer



Probing to the Viewers from Other Quartus II Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows within the Quartus II software. You can select one or more nodes or nets in another window and locate them in one of the viewers.

You can locate nodes between the RTL, State Machine, and Technology Map Viewers, and you can locate nodes in the RTL Viewer or Technology Map Viewer from the following Quartus II software windows:

- Project Navigator
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Node Finder
- Assignment Editor
- Messages Window
- Compilation Report
- TimeQuest Timing Analyzer (only supports the Technology Map Viewer)

To locate elements in the viewer from another Quartus II window, select the node or nodes in the appropriate window; for example, select an entity in the **Entity** list on the **Hierarchy** tab in the Project Navigator, or select nodes in the **Timing Closure Floorplan**, or select node names in the **From** or **To** column in the Assignment Editor. Next, right-click the selected object, point to Locate, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. After you choose this command, the viewer window opens, or is brought to the foreground if the viewer window is already open.

The first time the window opens after a compilation, the preprocessor stage runs before the viewer window opens.

The viewer shows the selected nodes and, if applicable, the connections between the nodes. The display is similar to what you see if you right-click the object, point to Filter, and click **Selected Nodes & Nets** using **Filter Across Hierarchy**. If the nodes cannot be found in the viewer, a message box displays the message: “Can’t find requested location.”

Viewing a Timing Path

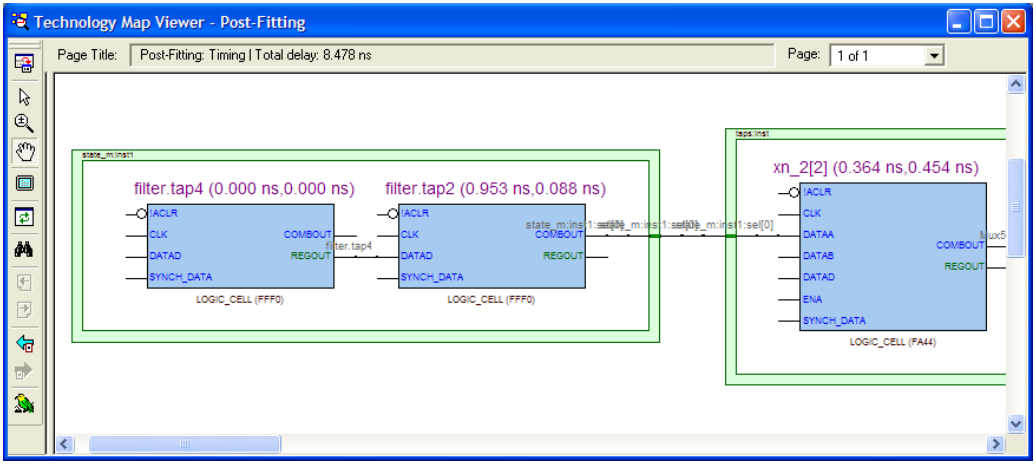
To see a visual representation of a timing path, you can cross-probe from the Timing Analysis section of the Compilation Report with the classic timing analyzer, or from a report panel in the TimeQuest timing analyzer.

To take advantage of this feature, you must first successfully complete a full compilation of your design, including the timing analyzer stage. To access the timing analyzer report that contains the timing results for your design, on the Processing menu, click **Compilation Report**. On the left side of the Compilation Report, select **Timing Analyzer** or **TimeQuest Timing Analyzer**. When you select a detailed report, the timing information is listed in a table format on the right side of the Compilation Report; each row of the table represents a timing path in the design. You can also view timing paths in TimeQuest report panels. To view a particular timing path in the Technology Map Viewer or the RTL Viewer, highlight the appropriate row in the table, right-click, point to Locate, and click **Locate in Technology Map Viewer** or **Locate in RTL Viewer**.

In the Technology Map Viewer, the schematic page displays the nodes along the timing path with a summary of the total delay. If you locate from the classic timing analyzer, the timing path also includes timing data representing the interconnect (IC) and cell delays associated with each node. The delay for each node is shown in the following format: <*post-synthesis node name*> (<*IC delay*> ns, <*cell delay*> ns). When you locate the timing path from the TimeQuest timing analyzer to the Technology Map Viewer, the interconnect and cell delay associated with each node is not displayed.

Figure 13–20 shows a portion of a classic timing analyzer timing path represented in the Technology Map Viewer. The total delay for the entire path through several levels of logic (only three levels are shown in Figure 13–20) is 7.159 ns. The delays are indicated for each level of logic. For example, the IC delay to the first `LCELL` primitive is 0.383 ns and the cell delay through the `LCELL` is 0.075 ns. When the timing path passes through a level of hierarchy, green hierarchy boxes group the logic and show the hierarchical boundaries. A green rectangular symbol on the border indicates the path passes between two different hierarchies.

Figure 13–20. Timing Path Schematic in the Technology Map Viewer



In the RTL Viewer, the schematic page displays the nodes in the path(s) 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 may 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 may 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 may display more than just the timing path. There are also some cases in which the path cannot be displayed, such as paths through state machines, encrypted intellectual property (IP), or registers that are created during the fitter process. In cases where the timing path displayed in the RTL Viewer might not be the correct path, the compiler issues messages.

Other Features in the Schematic Viewer

This section describes other features in the schematic view that enhance usability and help you analyze your design.

Tooltips

A tooltip is displayed whenever the mouse pointer is held over an element in the schematic. The tooltip contains useful information about a node, net, input port, and output port. Table 13–8 lists the information contained in the tooltip for each type of node.

The tooltip information for an instance (the first row in Table 13–8) includes a list of the primitives found within that level of hierarchy, and the number of each primitive contained in the current instance. The number includes all hierarchical blocks below the current instance in the hierarchy. This information lets you estimate the size and complexity of a hierarchical block without navigating into the block.

The tooltip information for atom primitives in the Technology Map Viewer (the second row of Table 13–8) shows the equation for the design atom. The equations are an expanded version of the equations you can view in the Equations window in the Timing Closure Floorplan. Advanced users can use these equations to analyze the design implementation in detail.

For details on understanding equations, refer to the Quartus II Help.

To copy tooltips into the clipboard for use in other applications, right-click the desired node or netlist, and click **Copy Tooltip**.

To turn off tooltips or change the duration of time that a tooltip is displayed in the view, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and set the desired options under **Tooltip settings**.

The **Show names in tooltip for** option specifies the number of seconds to display the names of assigned nodes and pins in a tooltip when the pointer is over the assigned nodes and pins. Selecting **Unlimited** displays the tooltip as long as the pointer remains over the node or pin. Selecting **0** turns off tooltips. The default value is 5 seconds.

The **Delay showing tooltip for** option specifies the number of seconds you must hold the mouse pointer over assigned nodes and pins before the tooltip displays the names of the assigned nodes and pins. Selecting **0**

displays the tooltip immediately when the pointer is over an assigned node or pin. Selecting **Unlimited** prevents tooltips from being displayed. The default value is 1 second.

Table 13–8. Tooltip Information (Part 1 of 2)

Description & Tooltip Format	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

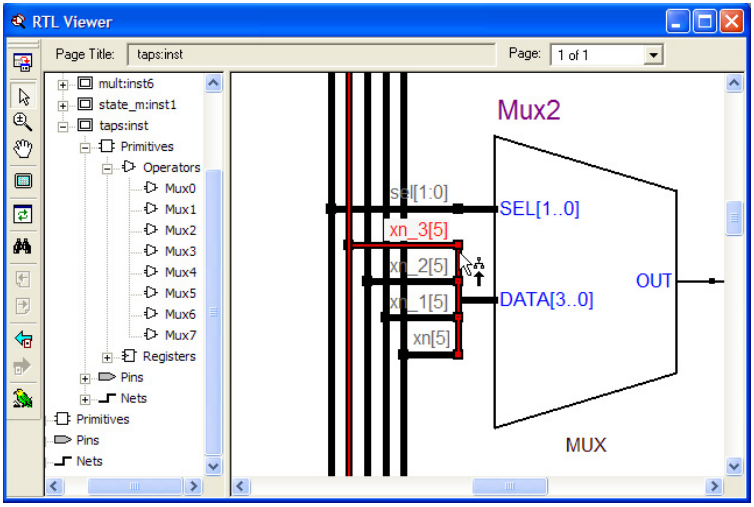
Table 13–8. Tooltip Information (Part 2 of 2)

Description & Tooltip Format	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).	Source from: reset:reset_i:rst *(1)* < Destination Index > | Source from: < [11] > | sample~0:OUT1 < [10] > | sample~1:OUT1 < [9] > | sample~2:OUT1 < [8] > | sample~3:OUT1 < [7] > | sample~4:OUT1 < [6] > | sample~5:OUT1 < [5] > | sample~6:OUT1 < [4] > | sample~7:OUT1 < [3] > | sample~8:OUT1 < [2] > | sample~9:OUT1 < [1] > | sample~10:OUT1 < [0] > | sample~11:OUT1 *(2)* < Destination Index > | Source from: < [7..6] > | node2:OUT1 < [5] > | ct[3]:OUT1 < [4] > | node2:OUT1 < [3..2] > | ct[3]:OUT1 < [1] > | node2:OUT1 < [0] > | ct[3]:OUT1 *(3)* < Destination Index > | Source from: < [11..0] > | 12' h000 *(4)* < Destination Index > | Source from: < [2..1] > | 2' h1 < [0] > | always7~2:OUT1 *(5)* < Destination Index > | Source from: < [15..0] > | md_me:inst18:dout[15:0] *(6)*
State Machine Node Format: <*node name*>	state_m:inst1|filter.tap1
State Machine Transition Arc This information is displayed when you hold your mouse over the arrow on the arc representing the transition between two states. Format: (<*equation for transition between states*>)	(!newt)

Rollover

You can highlight an element and view its name in your schematic using the rollover feature. When you place your mouse pointer over an object, the object is highlighted and the name is displayed, helping you to analyze your schematic diagram (Figure 13–21). This feature is enabled by default in the netlist viewers. To turn off the Rollover feature, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers** and turn off **Enable Rollover**.

Figure 13–21. Rollover in the RTL Viewer & Technology Map Viewer



The Properties Dialog Box

You can view the properties of an instance or a primitive using the Properties dialog box. To view the properties of an instance or a primitive in the RTL Viewer or the 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, VCC or GND). Table 13–9 describes the possible value of a port.

Table 13–9. Possible Port Values

Value	Description
VCC	The port is not connected and has VCC value (tied to VCC)
GND	The port is not connected and has GND value (tied to GND)
--	The port is connected and has value (other than VCC or GND)
Unconnected	The port is not connected and has no value (hanging)

Displaying Net Names

To see the names of all the nets displayed in your schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn on **Show Net Name** under **Display Settings**. This option is disabled by default. If you turn on this option, the schematic view refreshes automatically to display the net names.

Displaying Node Names

In some designs, nodes have long names that overlap the ports of other symbols in the schematic. To remove the node names from the schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn off **Show node name** under **Display Settings**. This option is turned on by default.

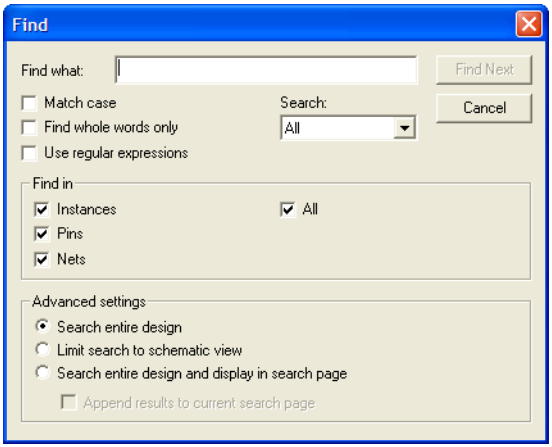
Full Screen View

To set the viewer window to fill the whole screen, on the View menu, click **Full Screen**, or click the **Full Screen** icon in the viewer toolbar, or press Ctrl+Alt+Space. The keyboard shortcut toggles the full screen. To return to the standard screen view after viewing the full screen, press Ctrl+Alt+Space again.

Find Command

To open the **Find** dialog box shown in Figure 13–22, on the Edit menu, click **Find**, or click the **Find** icon in the viewer toolbar, or right-click in the schematic view, and click **Find**.

Figure 13–22. Find Dialog Box



Select **Up** in the **Search** list to search from the current hierarchy to upper (parent) hierarchies. Select **Down** to search from the current hierarchy to lower (child) hierarchies. You can choose to search only instances (nodes) in the design, or to also search pins and nets. By default, only instances are searched.

When you click **Find**, the viewer selects and highlights the first item found, opens the appropriate page of the schematic, if necessary, and centers the page so that the node is visible in the viewable area (but does not zoom in to the node). To find the next matching node, click **Find Next**.

You can use the options in the **Advanced settings** section to control the scope of the results found during a search and how they are displayed in the viewer. The default selection, **Search entire design**, searches for the item in all design elements across the entire design. To search only in the pages of the currently displayed netlist, such as a schematic showing filtering results, choose **Limit search to schematic view**.

To display the results in a new page, select **Search entire design and display in search page**. This command searches all design elements across the entire design, and displays the results on a separate page dedicated to search results. You can also append new search results to an

existing search page with the **Append results to current search page** command. The appended items appear in the same relative position as they do in the full schematic. You can use this method to find and select two objects that are not on the same page and display them on the same page after performing the Find command.

Refer to "Finding Nodes in the RTL Viewer & Technology Map Viewer" in the Quartus II Help for more details about using the **Find** dialog box.

Exporting & Copying a Schematic Image

You can export the RTL Viewer or Technology Map Viewer schematic view in JPEG File Interchange Format (**.jpg**) or Windows Bitmap (**.bmp**) file format, which allows you to include the schematic in project documentation or share it with other project members. To export the schematic view, on the File menu, click **Export**. In the **Export** dialog box, type a file name and location, and select the desired file type. The default file name is based on the current instance name and the default file type is JPEG Interchange Format (**.jpg**). However, for pages that use filtering, expanding, or reducing operations, the default name is **Filter**<*number of export operation*>.<*file extension*>.

You can copy the whole image or only a portion of the image. To copy the full image, on the Edit menu, point to Copy and click **Full Image**. To copy a portion of the image, on the Edit menu, point to Copy and click **Partial Image**. The cursor changes to a plus sign to indicate that you can draw a box shape. Drag the cursor around the portion of the schematic you want to copy. When you release the mouse button, the partial image is copied to the clipboard.

Occasionally, due to the design size and objects selected, an image is too large to copy to the clipboard. In this case, the Quartus II software displays an error message.

To export or copy a schematic that is too large to copy in one piece, first split the design into multiple pages to export or to copy smaller portions of the design. For information about how to control how much of your design is shown on each schematic page, refer to "Partitioning the Schematic into Pages" on page 13–25. As an alterative, use the Partial Image feature to copy a portion of the image.

The **Copy** feature is not available on UNIX platforms.

Printing

To print your schematic page, on the File menu, click **Print**. You can print each schematic page onto one full page, or you can print the selected parts of your schematic onto one page with the **Selection** option. Refer to "Partitioning the Schematic into Pages" on page 13–25 to control how much of your design is shown on each schematic page.

Before printing, you can modify the page orientation. On the File menu, click **Page Setup**. Change the page orientation from **Portrait** to **Landscape**, or to the setting that best fits your design. You also can adjust the page margins in the **Page Setup** dialog box.

The hierarchy list in the viewers and the table view of the State Machine Viewer cannot be printed. You can use the State Machine Viewer **Copy** command to copy the table to a text editor and print from the text editor.

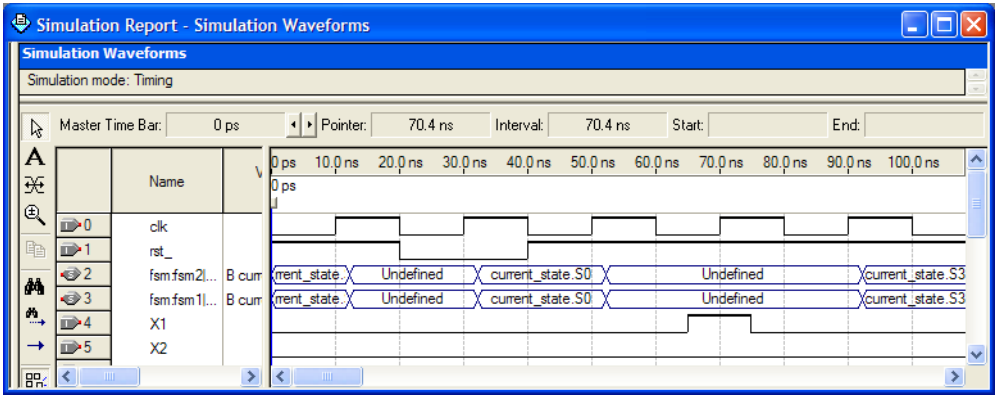
Debugging HDL Code with the State Machine Viewer

This section provides an example of using the State Machine Viewer to help debug HDL code. This example shows how you can use the various features in the netlist viewers to help solve design problems.

Simulation of State Machine Gives Unexpected Results

This section presents a design scenario in which you compiled your design and performed a simulation in the Quartus II Simulator. The simulation result is shown in Figure 13–23 and has unexpected undefined states.

Figure 13–23. Simulation Result Showing Undefined States



To analyze the state machine design in the State Machine Viewer, follow these steps:

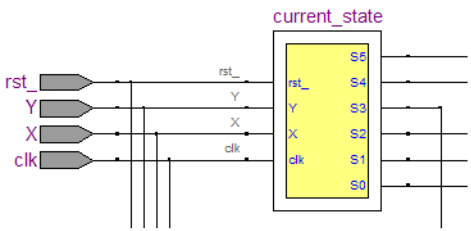
1. Open the State Machine Viewer for the state machine of interest. You can do this in any of the following ways:

   - On the Tools menu, point to Netlist Viewers and click **State Machine Viewer**. In the State Machine selection box, choose the state machine that you want to view.

   - On the Tools menu, point to Netlist Viewers, and click **RTL Viewer**. Browse to the hierarchy block that contains the state machine definition and double-click the yellow state machine instance to open the State Machine Viewer (Figure 13–24).You can open the State Machine Viewer using either of two methods:

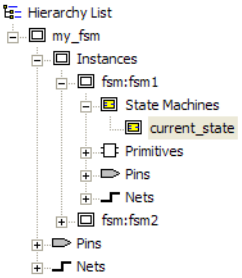
     - In the schematic view, double-click an instance in the hierarchy to open the lower level hierarchy. You can traverse through the schematic hierarchy in this way to open the schematic page that contains the state machine (Figure 13–24).

Figure 13–24. State Machine Instance in RTL Viewer Schematic View



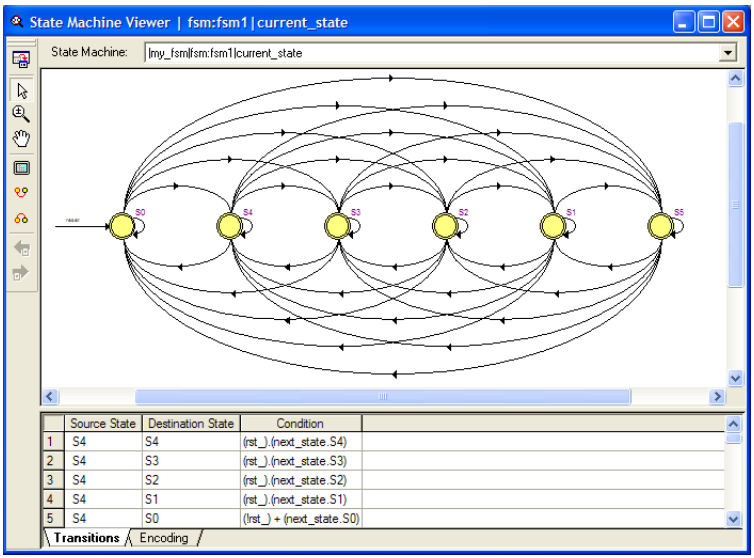
   - In the hierarchy list, click the + symbol next to **Instances** to open a list of the instances in that hierarchy level of the design. You can traverse down the hierarchy tree in this way to find the instance that contains the state machine. Click on the name of the state machine in the **State Machines** folder (Figure 13–25) to open the appropriate schematic in the schematic view (Figure 13–24).

Figure 13–25. State Machine Instance in RTL Viewer Hierarchy List



The State Machine Viewer opens (Figure 13–26).

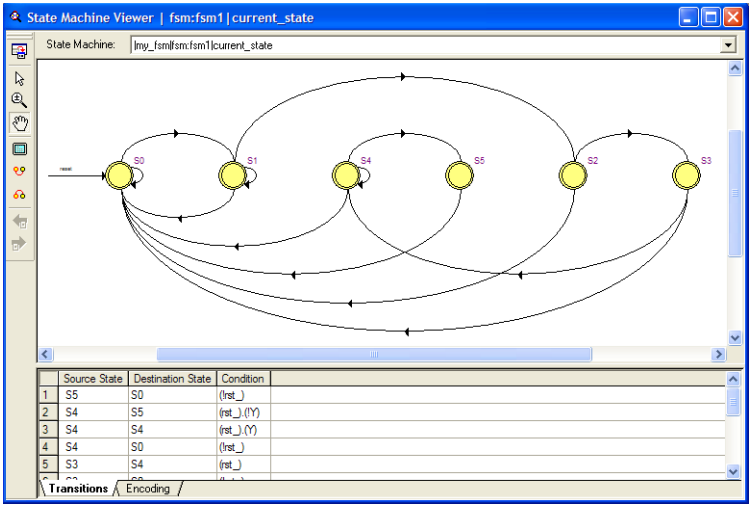
Figure 13–26. State Machine Viewer Showing Incorrect Transitions



2. You can analyze this state machine instance using the state machine diagram, transition table, and encoding table. Clearly something is wrong with the state machine because every state has a transition to every other state. After inspecting the state machine behavior, you determine that in this scenario, the designer forgot to create default assignments for the next state (that is, `next_state = current_state` if the conditions are not met).

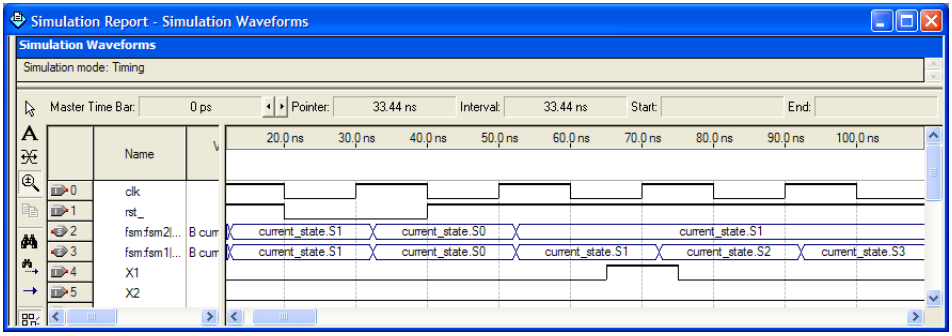
3. After fixing the error in the HDL code, recompile the design and repeat steps 1-2 to view the new state machine diagram and transition table shown in Figure 13–27 and check that the state transitions now occur correctly.

Figure 13–27. State Machine Viewer Showing Correct Transitions



4. Perform a new simulation, as shown in Figure 13–28, and verify that the state machine now performs as expected.

Figure 13–28. Simulation Result Showing Correct States



Conclusion

The Quartus II RTL Viewer, State Machine Viewer, and Technology Map Viewer allow you to explore and analyze your initial synthesis netlist, post-synthesis netlist, or post-fitting and physical synthesis netlist. The viewers provide a number of features in the hierarchy list and schematic view to help you quickly trace through your netlist and find specific hierarchies or nodes of interest. These capabilities can help you debug, optimize, or constrain your design more efficiently to increase your productivity.

Document Revision History

Table 13–10 shows the revision history for this document.

Table 13–10. Documentation Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Chapter 13 was formerly Chapter 12 in version 6.0.0. Updated for the Quartus II software version 6.1.0: • Added information about the Technology Map Viewer (Post-Mapping) • Can run the RTL Viewer as part of compilation flow, rather than wait for the Fitter to complete before viewing the netlist • Customized the schematic display for better viewing and to speed up the debugging process • Added support for Stratix III devices	With the addition of the Technology Map Viewer (Post-Mapping), you can view both the post-mapping and post-fitting netlists at the same time. Other changes also speed up the debugging process.
May 2006 v6.0.0	Name changed to *Analyzing Designs with the Quartus II Netlist Viewers*. Updated for the Quartus II software version 6.0.0: • Updated GUI information.	
December 2005 v5.1.1	Updated for version 5.1, including viewing inside device atoms, filter on bus index, display timing path in the RTL Viewer, state machine access from Tools menu, locate from state machines, and state encoding table.	
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 12 was formerly chapter 14 in version 5.0.	
May 2005 v5.0.0	Chapter 14 was formerly chapter 12 in version 4.2.	
December 2004 v2.1	• Chapter 13 was formerly Chapter 14 in version 4.1. • Updates to tables and figures. • New functionality for Quartus II software version 4.2.	
June 2004 v 2.0	• Updates to tables, and figures. • New functionality for Quartus II software version 4.1.	
February 2004 v1.0	Initial release.	