AN 307: Altera Design Flow for Xilinx Users

AN-307-7.0 Application Note

Introduction

Designing for Altera® Field Programmable Gate Array devices (FPGAs) is very similar, in concept and practice, to designing for Xilinx FPGAs. In most cases, you can simply import your register transfer level (RTL) into Altera's Quartus® II software and begin compiling your design to the target device.

This document is intended for Xilinx designers who are familiar with the Xilinx ISE software and would like to convert their existing ISE designs to the Altera Quartus II software environment.

The first section of this application note, "Quartus II Approach to FPGA Design", shows the equivalent flows between the Altera Quartus II software and the Xilinx ISE software. Comparisons between Xilinx ISE software and Altera Quartus II software are also made whenever possible.

The second section of this application note, "Xilinx to Altera Design Conversion", provides ISE design guidelines for conversion to the Altera Quartus II software. It also discusses the similarity and equivalent features of the ISE and the Quartus II software, including Xilinx CORE generator modules and instantiated primitives. In addition, this application note demonstrates the device and design constraints conversion from Xilinx ISE software to Altera Quartus II software.

This application note is based on the latest information available for the Quartus II software version 12.1 and Xilinx ISE software version 14.2.

For more information about setting up your design in the Quartus II software, refer to the *Quick Start Guide For Quartus II Software*.

Quartus II Approach to FPGA Design

The Quartus II software provides a suite of tools similar to those found in the Xilinx ISE software. The Quartus II software allows you to perform design implementation either by using command-line executables and scripting, or by using the Quartus II GUI.

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

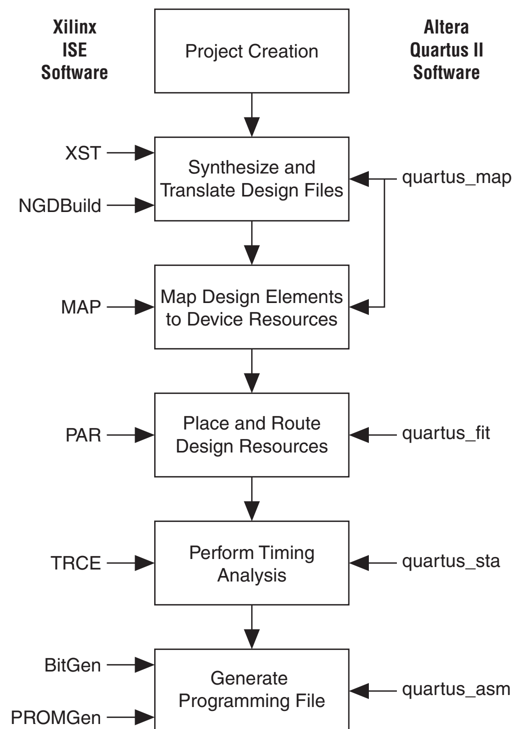
© 2013 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

FPGA Design Flow Using Command Line Scripting

The ability to automate the FPGA design process saves time and increases productivity. The ISE software and the Quartus II software provide the tools necessary to automate your FPGA design flow. Figure 1 shows the similarity between a typical command line implementation flow using either the Xilinx ISE software or the Altera Quartus II software.

Figure 1. Basic Design Flow



For more information about the Quartus II command-line executable flow, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

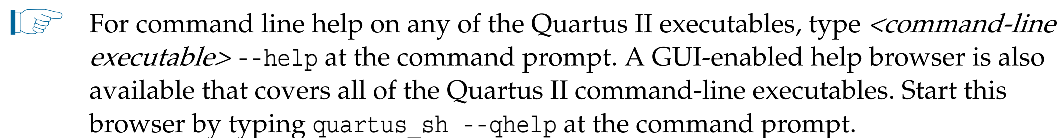
For ISE software users who are familiar with the command-line implementation flow that compiles a design and generates programming files for FPGA design files, a similar flow exists in the Quartus II software, known as the compilation flow. The compilation flow is the sequence and methods by which the Quartus II software translates your design files, maps the translated design to device-specific elements, places and routes the design in the device, and generates a programming file. These functions are performed by the Quartus II Integrated Synthesis (QIS), Fitter, Assembler, and TimeQuest timing analyzer. The following sections describe and compare the two software flows using command line executables.

Command-Line Executable Equivalents

Table 1 summarizes the most common executable equivalents between the ISE software and the Quartus II software.

Table 1. Command-Line Executable Equivalents

Xilinx ISE Executable	Altera Quartus II Executable	Description
XST NGDBuild MAP	quartus_map	Translates project design files (for example, RTL or EDA netlist), and maps design elements to device resources.
PAR	quartus_fit	Places and routes the device resources into the FPGA.
TRCE	quartus_sta	Performs a static timing analysis on the design.
BitGen PROMGen	quartus_asm	Generates programming file from post-place-and-route design.
NetGen	quartus_eda	Generates output netlist files for use with other EDA tools.
XPWR	quartus_pow	Performs power estimation on the design.
XFLOW	quartus_sh -flow quartus_cdb	Automates synthesis, implementation, and simulation flows. In addition, the quartus_cdb executable allows you to import and export version-compatible databases and merges partitions.



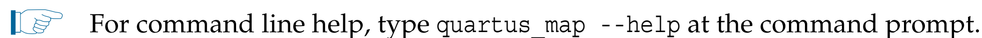
For command line help on any of the Quartus II executables, type *<command-line executable>* --help at the command prompt. A GUI-enabled help browser is also available that covers all of the Quartus II command-line executables. Start this browser by typing quartus_sh --qhelp at the command prompt.

XST/NGDBuild/MAP

In the Quartus II command-line flow, only one executable (that is, quartus_map) is required to perform synthesis (XST), the design translation (NGDBuild) and mapping of design elements to device resources (MAP).

The following example runs logic synthesis and technology mapping of your design (**filtref.v**) targeting the Stratix® V device family:

```
quartus_map filtref --rev=<revision name> --source=filtref.v --family=stratixv
```



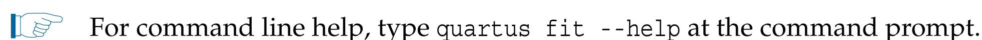
For command line help, type quartus_map --help at the command prompt.

PAR

In place of the PAR executable provided by the ISE software to place and route all device resources into your selected FPGA device, the Quartus II software provides the quartus_fit command-line executable.

The following example performs place-and-route by fitting the logic of your Quartus II project, filtref, into your target Stratix V 5SGXEA7K2F40C2 device:

```
quartus_fit filtref --part=5SGXEA7K2F40C2
```



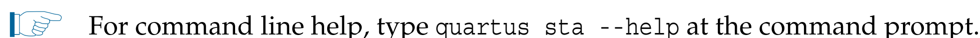
For command line help, type quartus_fit --help at the command prompt.

TRCE

In place of the `TRCE` executable provided in the ISE software for performing a static timing analysis on your design, the Quartus II software provides the `quartus_sta` executable. The Quartus II software uses the industry standard Synopsis Design Constraint (SDC) file format for timing constraints rather than the User Constraint File (**.ucf**) constraint format created by Xilinx.

The following example performs timing analysis on the filtref project with the SDC timing constraints file, **filtref.sdc**, to determine whether the design meets the timing requirements:

```
quartus_sta filtref --sdc=filtref.sdc
```


For command line help, type `quartus_sta --help` at the command prompt.

For more information about creating timing constraints, refer to the section "TimeQuest Timing Analyzer SDC Editor" on page 13.

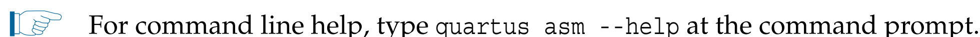
BitGen/PROMGen

The ISE software provides the `BitGen` and `PROMGen` executables to generate FPGA programming files. The Quartus II software provides the `quartus_asm` executable to generate programming files for FPGA configuration.

The following example creates programming files (for example, **filtref.sof** or **filter.pof**) for the filtref project:

```
quartus_asm filtref
```

The programing files are used to program and configure the FPGA.


For command line help, type `quartus_asm --help` at the command prompt.

NetGen

The `NetGen` executable reads Xilinx design files as input, extracts data from the design files, and generates netlists that are used with supported third-party tools (for example, simulation and static timing analysis). Similarly, the Quartus II software provides the `quartus_eda` executable to generate netlists and other output files for use with third-party EDA tools.

The following example creates simulation Verilog HDL netlist files (**filtref.vo** and **filtref.sdo**) for the filtref project to run with ModelSim®:

```
quartus_eda filtref --simulation=on --format=verilog --tool=modelsim
```

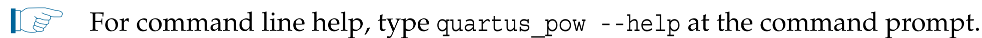
For command line help, type `quartus_eda --help` at the command prompt.

XPWR

The `XPWR` executable provides power and thermal estimates after `PAR` to estimate your design's power consumption. Similarly, `quartus_pow` provides the thermal dynamic and thermal static power consumed by your design.

The following example uses the **filtref.vcd** file as input to perform power analysis and perform glitch filtering on the VCD file:

```
quartus_pow filtref --input_vcd=filtref.vcd --vcd_filter_glitches=on
```

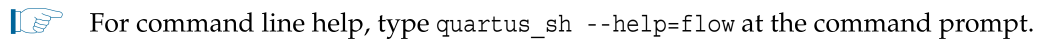

For command line help, type `quartus_pow --help` at the command prompt.

XFLOW

Similar to the ISE software's `XFLOW` implementation command, the Quartus II shell (`quartus_sh`) provides a `--flow` option that you can use to perform a complete compilation for your design project.

The following example runs compilation, timing analysis, and programming file generation with a single command:

```
quartus_sh --flow compile <project name>
```


For command line help, type `quartus_sh --help=flow` at the command prompt.

In addition, the **quartus_cdb** executable allows you to import and export version-compatible databases. This is useful if you want to migrate your existing design between different versions of the Quartus II software. To do so, you can export a database from one version of the Quartus II software and import it directly into another version of the software. This eliminates unnecessary compilation time, and you only need to rerun the timing analysis or simulation with the updated timing models. Full compilation is not required.

In addition, the **quartus_cdb** executable allows you to merge partitions for incremental compilation.

For more information about incremental compilation, refer to the section "Quartus II Incremental Compilation" on page 24.

Programing and Configuration File Support in the Quartus II Software

The Quartus II software requires different programming and configuration files based on the type of device and configuration mode.

Table 2 lists the programming and configuration file formats supported by Altera FPGAs, CPLDs, and configuration devices.

Table 2. Programming and Configuration File Format

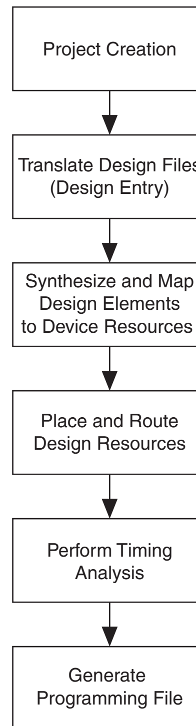
File Format	FPGA	CPLD	Configuration Device	Serial Configuration Device
SRAM Object File (**.sof**)	✓	—	—	—
Programmer Object File (**.pof**)	—	✓	✓	✓
JEDEC JESD71 STAPL Format File (**.jam**)	✓	✓	✓	—
Jam Byte Code File (**.jbc**)	✓	✓	✓	—

For more information on programming and configuring devices with the Quartus II software, refer to the *Quartus II Programmer* chapter in volume 3 of the *Quartus II Handbook*.

FPGA Design Flow Using Tools with GUIs

The Quartus II and ISE software GUIs address each of the major FPGA design steps shown in Figure 2 in different ways. The following subsections present the Altera equivalents for Xilinx ISE GUI features.

Figure 2. Typical FPGA Design Flow



GUI Feature Equivalents

Table 3 summarizes the most common GUI feature equivalents between the ISE software and Quartus II software.

Table 3. GUI Feature Equivalents (Part 1 of 2)

GUI Feature	Xilinx ISE Software	Altera Quartus II Software
Project Creation	New Project	New Project Wizard
Design Entry	HDL Editor	HDL Editor
	EDA Netlist	EDA Netlist Design Entry
	Schematic Editor (2)	Schematic Editor
	State Diagram Editor (StateCAD) (3)	State Machine Editor
	CoreGen & Architecture Wizard	MegaWizard™ Plug-In Manager
	Xilinx Platform Studio and Embedded Development Kit (3)	Qsys system integration tool
Design Constraints	Xilinx Constraints Editor	Assignment Editor, Quartus II TimeQuest timing analyzer SDC Editor
	Floorplan I/O Editor, PinAhead Technology	Pin Planner

Table 3. GUI Feature Equivalents (Part 2 of 2)

GUI Feature	Xilinx ISE Software	Altera Quartus II Software
Synthesis	Xilinx Synthesis Technology (XST)	Quartus II Integrated Synthesis (QIS)
	Third-Party EDA Synthesis	Third-Party EDA Synthesis
Design implementation	Translate, Map, Place & Route	Quartus II Integrated Synthesis (QIS), Fitter
Static timing analysis	Xilinx Timing Analyzer	TimeQuest timing analyzer
Generation of device programming files	BitGen	Assembler
Power analysis	XPower	PowerPlay Power Analyzer
Simulation	ModelSim Xilinx Edition (1)	ModelSim-Altera Starter Edition (5)
	Third-Party Simulation Tools	Third-Party Simulation Tools
	ISE Simulator (1)	
Hardware verification	ChipScope Pro (1)	SignalTap II Logic Analyzer
	Virtual IO	In-System Sources and Probes Editor
Viewing and editing design placement	FloorPlan Area/Logic Editor, FPGA Editor	Chip Planner
Technique to improve productivity and optimize design	SmartCompile	Quartus II Incremental Compilation
	PlanAhead (1)	Physical synthesis optimization
	SmartXplorer (4)	Design Space Explorer (DSE)

Notes to Table 3:

(1) Feature that requires additional cost, but is included in the Quartus II software.

(2) Supported in MS Windows only, but the Quartus II software supports all platforms.

(3) Supported in MS Windows and Linux only, but the Quartus II software supports all platforms.

(4) Supported in Linux only, but the Quartus II software supports all platforms.

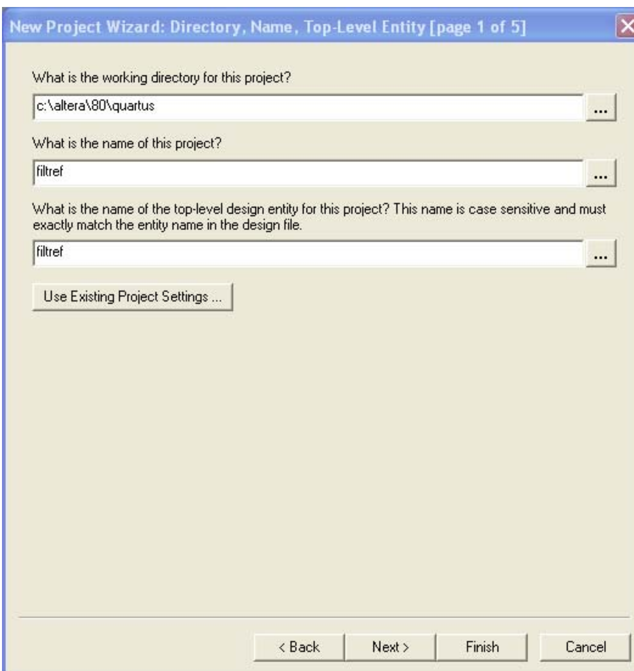
(5) This feature is provided for free when you purchase the Quartus II software.

For more information about the Quartus II software, refer to the *Introduction to the Quartus II Software* manual.

Project Creation

To begin a design in either the ISE or Quartus II software, you must first create a project. The project includes all design files and compiler settings that control the compilation and optimization of the design during synthesis and fitting. Similar to the **New Project** command on the File menu in the ISE software, the Quartus II software uses the New Project Wizard to guide you through the steps of specifying a project name and directory, the top-level design entity, any EDA tools you are using, and a target device. To invoke the New Project Wizard, on the File menu, select **New Project Wizard**. Figure 3 shows page 1 of the Quartus II New Project Wizard.

Figure 3. The New Project Wizard Page 1



After creating a new project, the Quartus II software automatically generates project files necessary for successful compilation, including the Quartus II Project File (**.qpf**) and Quartus II Settings File (**.qsf**).

- The Quartus II Project File (**.qpf**) contains basic information about the current version of the Quartus II software, the date, and also lists all of the revisions created for the project.
- The Quartus II Settings File (**.qsf**) contains all of the project-wide and entity-level assignments and settings for the current revision of the project. A separate **.qsf** exists for each individual revision.
- All the settings that you make when creating your project with the New Project Wizard can be modified later in the design process.

Design Entry

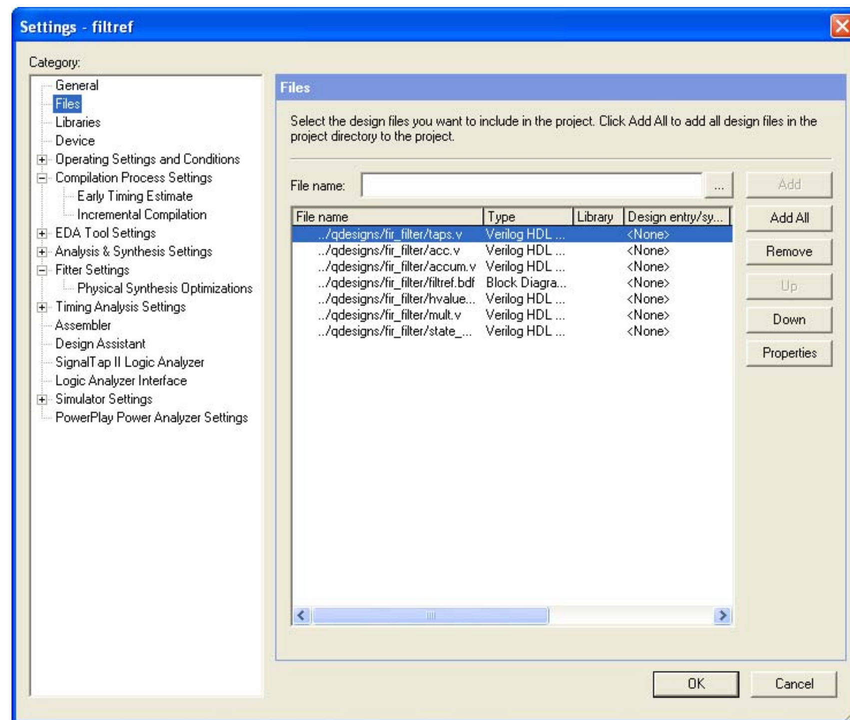
The ISE software and the Quartus II software support hardware description language (HDL), EDA netlist, and schematic design files as design entry methods.

In the Quartus II software, to add or remove existing design files from your project, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Files**. The **Files** page appears, allowing you to add or remove files (Figure 4).

In the Xilinx ISE software, existing design files are added or removed in the **Add Source** dialog box.

Figure 4. Files Page of the Settings Dialog Box



HDL Editor

Similar to the New Source Wizard on the Project menu of the ISE software, to create a new HDL design file in the Quartus II software, on the File menu, click **New** and select the type of file you want to create. To assist you in creating HDL designs, the Quartus II software provides design example templates for VHDL and Verilog HDL, including language constructs examples to help you get started on your design. Also, the Quartus II Text Editor offers syntax coloring for highlighting HDL reserved words and comments.

For more information about design guidelines in the Quartus II software, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* and *Recommended HDL Coding Styles* chapters in volume 1 of the *Quartus II Handbook*.

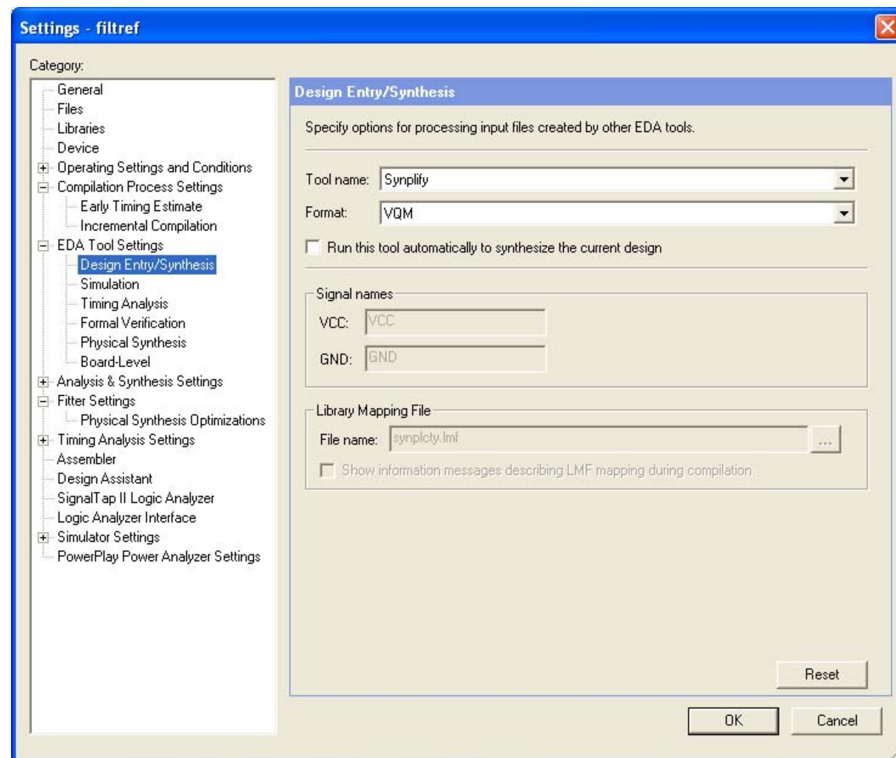
EDA Netlist Design Entry

Both ISE and Quartus II software allow you to compile designs from the netlists generated from third-party EDA tools such as Synopsys Synplify or Mentor Graphics® Precision RTL. To specify the third-party synthesis tools in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, expand the "+" icon under **EDA Tool Settings** and select **Design Entry/Synthesis**. The **Design Entry/Synthesis** page appears. You can also specify this in the New Project Wizard.

Figure 5 shows the **Design Entry/Synthesis** page.

Figure 5. Design Entry/Synthesis Page



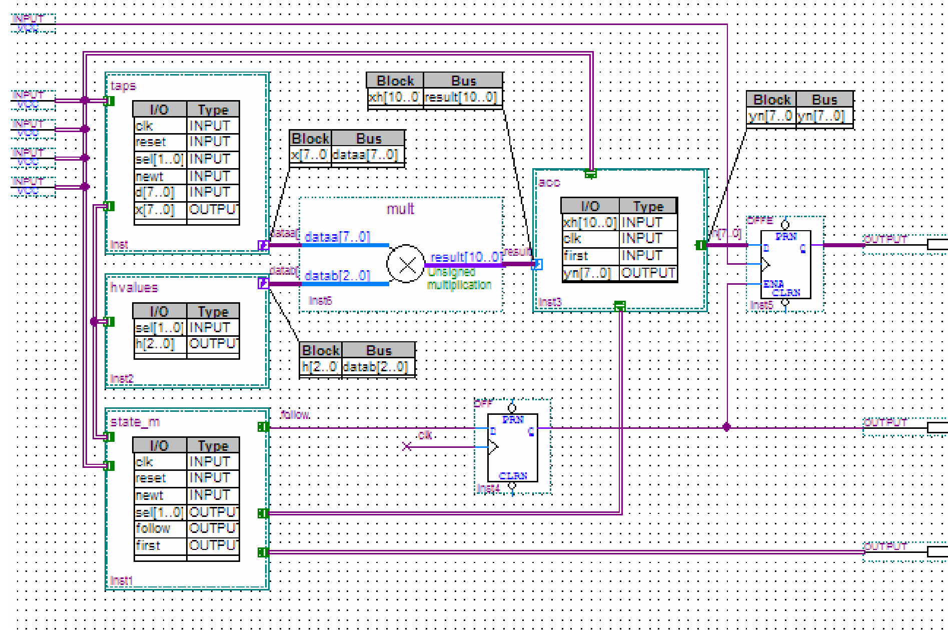
For more information about using third-party synthesis tools, refer to the synthesis chapters in *Volume 1* of the *Quartus II Handbook*.

Schematic Editor

In the Quartus II software, you can use Altera-supplied design elements, such as Boolean gates and registers, or you can create your own symbols from HDL or EDA netlist design entities. To create a block design file from a VHDL design file, a Verilog HDL design file, or an EDA netlist, on the File menu, point to **Create/Update** and click **Create Symbol Files for Current File**.

Similar to the New Source Wizard in the Project menu of the ISE software, to create a new schematic file (***.bdf**) in the Quartus II software, on the File menu, click **New** and select the Block Diagram/Schematic file. To insert block symbols into the schematic, double-click the schematic file and choose the appropriate block symbols. Figure 6 shows an example of the schematic file.

Figure 6. Schematic File



State Machine Editor

In place of the State Diagram Editor (StateCAD) in the ISE software, the State Machine Editor in the Quartus II software allows you to create graphic representations of state machines for use in your design. You can generate a Verilog Design File (**.v**) or VHDL Design File (**.vhd**) directly from the State Machine Editor with the **Generate HDL File** command.

Similar to the New Source Wizard in the Project menu of the ISE software, to create a new **.bdf** file in the Quartus II software, on the File menu, click **New** and select the State Machine file.

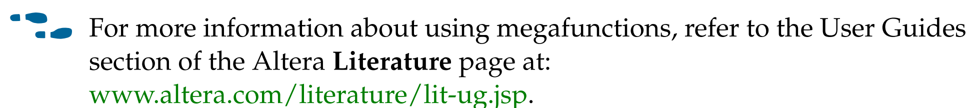
For more information about using the State Machine Editor, refer to the *Creating and Editing State Machines with the State Machine Editor* in the Quartus II Help.

MegaWizard Plug-In Manager

In place of the Core Generator and the Architecture Wizard available in the Xilinx ISE software, the Altera MegaWizard Plug-In Manager helps you to create highly customized megafunctions that are optimized for the device targeted in your design. These customizations draw on Altera-provided megafunctions, including library-of parameterized-modules (LPM) functions, ranging from simple Boolean gates to complex memory structures. The MegaWizard Plug-In Manager categorizes all supported modules into folders titled: Arithmetic, Gates, I/O, Memory Compiler, Storage, and others.

You can access the MegaWizard Plug-In Manager as a stand-alone tool or as an integrated tool in your Quartus II software. Use the MegaWizard Plug-In Manager to generate Altera equivalents for Xilinx primitives and CoreGen and Architecture Wizard modules.

The MegaWizard Plug-In Manager also serves as a convenient way to access and instantiate Altera IP cores.

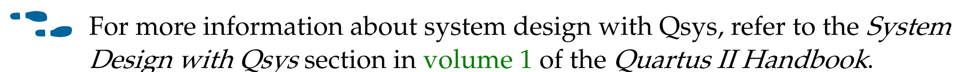


For more information about using megafunctions, refer to the User Guides section of the Altera **Literature** page at: www.altera.com/literature/lit-ug.jsp.

Qsys System Integration Tool

In place of the Xilinx Platform Studio and Embedded Development Kit (EDK) available in the Xilinx ISE software, the Qsys System Integration tool (Qsys) in the Quartus II software enables the use of processors (such as the Altera Nios® II embedded processor), interfaces to off-chip processors, standard peripherals, IP cores, on-chip memory, off-chip memory, and user-defined logic into a custom system module.

Qsys generates a single system module that instantiates these components and automatically generates the necessary interconnect logic to bind them together.



For more information about system design with Qsys, refer to the *System Design with Qsys* section in volume 1 of the *Quartus II Handbook*.

Design Constraints

Specifying device and timing constraints assures that your design takes advantage of specific features of your targeted device architecture and meets performance goals. The ISE software provides the Constraints Editor, Floorplan I/O Editor, and PinAhead Technology to create and edit constraints.

In the Quartus II software, after you have created your design, the Quartus II Assignment Editor and SDC Editor in the TimeQuest timing analyzer conveniently allow you to create and view constraints such as pin assignments, device options, logic options, and timing constraints. In place of the Floorplan I/O Editor and PinAhead Technology, the Quartus II Pin Planner allows you to view, create, and edit pin assignments in a graphical interface.

Table 4 summarizes the file format and assignment types that are set by the tools in the Quartus II software.

Table 4. Quartus II Assignment Tools

Assignment Type	File Format	Tools to Make Assignments
Timing	SDC	TimeQuest timing analyzer
I/O-related	QSF	Pin Planner, Assignment Editor
Others		Assignment Editor

Assignment Editor

The Quartus II Assignment Editor on the Assignments menu allows you to make timing and placement design constraints for your design. The Quartus II software dynamically validates the assignments whenever changes are made with the Assignment Editor, and issues errors or warnings for invalid assignments. Adding or changing assignments is acknowledged with messages reported in the **System** tab of the Quartus II message utility window. Figure 7 shows the Quartus II Assignment Editor, which is launched directly from the Assignments menu.

Figure 7. Quartus II Assignment Editor

Assignment Editor*

<<new>> Filter on node names: * Category: All

	Status	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	Ok		clk	Location	IOCONF...11_N21	Yes			
2	Missing Value			Location		Yes			
3		<<new>>	<<new>>	<<new>>					

For more information about the Quartus II Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

TimeQuest Timing Analyzer SDC Editor

The SDC file is an industry-standard Synopsys Design Constraint (SDC) methodology for constraining timing in designs. The Quartus II TimeQuest timing analyzer allows you to conveniently modify and create the timing constraints through a GUI interface.

Your Quartus II project must be open before using the TimeQuest timing analyzer. After the project is open, use the TimeQuest timing analyzer's GUI or SDC editor to create your design's SDC file, or to add an existing SDC file to your current project.

To create a new SDC file, on the File menu, select **New SDC File**. The Quartus II software provides templates for SDC constraints you can apply by clicking on **TimeQuest** in the **Insert Template** command. The **TimeQuest** folder contains templates for functions such as collections, clocks, clock attributes, I/O delays and common exceptions.

For more information and templates for specific constraints, refer to the *Quartus II TimeQuest Timing Analyzer Cookbook*.

The TimeQuest SDC Editor offers syntax coloring for highlighting SDC reserved words and comments. A tooltip also appears that shows the options and format for the constraint or exception. Figure 8 shows an example of an SDC in the Text Editor in the TimeQuest timing analyzer.

Figure 8. SDC Editor in the TimeQuest Timing Analyzer


```
13  # WARNING: Expected ENABLE_CLOCK_LATENCY to be set to 'ON', but it is set to 'OFF'
14  #          In SDC, create_generated_clock auto-generates clock latency
15  #
16  # ------------------------------------------
17  #
18  # Create generated clocks based on PLLs
19  derive_pll_clocks -use_tan_name
20  #
21  # ------------------------------------------
22  # WARNING: Global Fmax translated to derive_clocks. Behavior is not identical
23  if {![info exist ::qsta_message_posted]} {
24      post_message -type warning "Original Global Fmax translated from QSF using derive_clocks"
25      set ::qsta_message_posted 1
26  }
27  derive_clocks -period "85 MHz"
28  #
29
30
31  # Original Clock Setting Name: clk
32  create_clock -period "11.764 ns" \
33               -name {clk} {clk}
34  # ---------------------------------------------
35
36
37  # Original Clock Setting Name: clkx2
38  create_clock -period "11.764 ns" \
39               -name {clkx2} {clkx2}
40
41
42  # ---------------------------------------------
43  # The following clock group is added to try to
44  # match the behavior of:
45  #   CUT_OFF_PATHS_BETWEEN_CLOCK_DOMAINS = ON
46  # ---------------------------------------------
47
48  set_clock_groups -asynchronous \
49                  -group { \
50                        clkx2 \
51                         } \
52                  -group { \
53                        clk \
54                         } \
55
56  # ---------------------------------------------
```

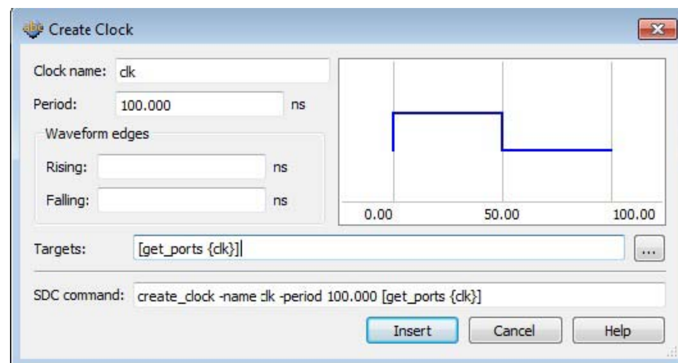

To use the TimeQuest timing analyzer GUI to create timing constraints, you must first create a timing netlist. On the Netlist menu, click **Create Timing Netlist**. After the timing netlist is created, use the Constraints menu in the TimeQuest timing analyzer GUI to constrain your design.

The following constraints are available on the Constraint menu:

- Create Clock
- Create Generated Clock
- Set Clock Latency
- Set Clock Uncertainty
- Set Input Delay
- Set Output Delay
- Set False Path
- Set Multicycle Path
- Set Maximum Delay
- Set Minimum Delay

For example, when you click **Create Clock**, the **Create Clock** dialog box appears (Figure 9), which allows you to set the clock constraints for your designs.

Figure 9. Create Clock Constraint through the Constraint Menu in the TimeQuest Timing Analyzer



After entering the values in the dialog box click **Insert** to insert the SDC command into your SDC file. Save your updated SDC file with the **Save** or **Save As** command from the File menu. You can also **Open** an existing SDC in the Text Editor in TimeQuest from the File menu.

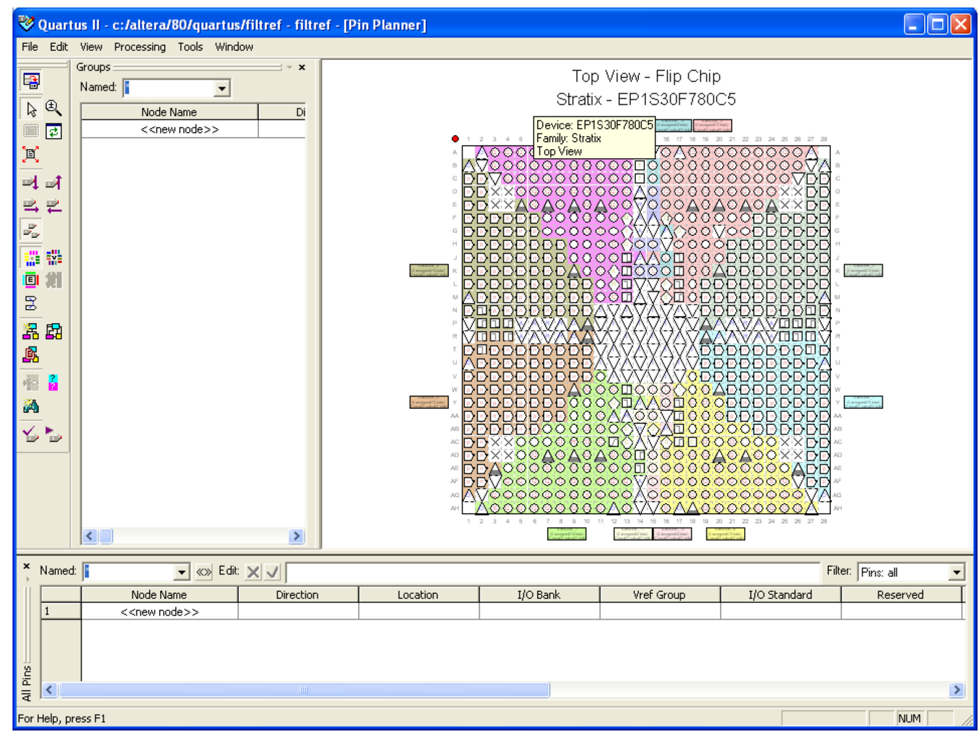
For more information about using the SDC file with the TimeQuest timing analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in the *Quartus II Handbook*.

For guidelines on timing constraints used by the Quartus II TimeQuest Timing Analyzer refer to *Specifying Timing Constraints and Exceptions*. in Quartus II Help.

Pin Planner

Similar to the Floorplan I/O Editor in the ISE software, the Quartus II Pin Planner provides a graphical Package view, allowing you to validate your I/O assignments by performing legality checks on your design I/O's pins and surrounding logic. With the Pin Planner, you can identify I/O banks, VREF groups, and differential pin pairings to help you with the I/O planning process. To use the Pin Planner, on the Assignments menu, click **Pin Planner**. Figure 10 shows the Pin Planner.

Figure 10. Quartus II Pin Planner



For more information about using the Quartus II Pin Planner, refer to the *I/O Management* chapter in the *Quartus II Handbook*.

Synthesis

Similar to the Xilinx Synthesis Technology (XST) in the ISE software, the Quartus II software includes Quartus II Integrated Synthesis (QIS), which provides full synthesis support for VHDL, Verilog HDL, and SystemVerilog as well as Altera-specific AHDL and Block Design File (**.bdf**) schematic entry. The integrated synthesis engine is the default engine.

The **Analysis & Synthesis Settings** dialog box allows you to set options that affect the analysis and synthesis stage of the compilation flow. These options include Optimization Technique, State Machine Processing, Restructure Multiplexers, and others.

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

The Quartus II software also supports synthesized design files from third-party synthesis tools including EDIF (**.edf**) and Verilog Quartus II Mapping (**.vqm**) netlist files. To specify which third-party EDA synthesis to use, refer to the section "EDA Netlist Design Entry" on page 9.

For more information about using third-party synthesis tools, refer to the synthesis chapters in *Volume 1* of the *Quartus II Handbook*.

Design Implementation

The ISE software follows an implementation flow that compiles a design and generates a programming file for your FPGA design files. A similar flow exists in the Quartus II software, and is known as the compilation flow.

The compilation flow is the sequence and method by which the Quartus II software translates your design files, maps the translated design to device specific elements, performs place and route for the design in the device, and generates a programming file. These functions are performed by Analysis and Synthesis, Fitter, and Assembler.

You can start the compilation flow at any point in the design process, whether or not you have completed making your project settings and constraints. In the Quartus II software, on the Processing menu, click **Start Compilation** to start the compilation process.

In the initial compilation phase, Analysis and Synthesis creates a database from your design files that contains all necessary design information. A design rule check is performed on all design files in the project, ensuring that no boundary connectivity errors or syntax errors exist. This database is available for use in all subsequent steps in the compilation flow.

Analysis and Synthesis optimizes your design for the targeted Altera FPGA and maps the design to the device. Mapping converts your design files into architecture-specific atoms that target device resources, such as logic elements (LEs) and RAM blocks.

The Fitter places and routes the atoms created by Analysis and Synthesis in the selected device. The Fitter performs additional optimization to improve your design's timing and resource usage based on timing constraints.

When the optimal fit is achieved, the Assembler generates the programming file for your design. The programming file contains all placement and routing information of your design and is ready to be programmed to your target Altera device.

The Tasks window shows the progress of the current compilation (Figure 11). The results of a compilation are viewed in the Compilation Report window. The window opens automatically when you compile a design, and shows the design hierarchy, a compilation summary, and statistics on the performance of the design.

Figure 11. Tasks Window

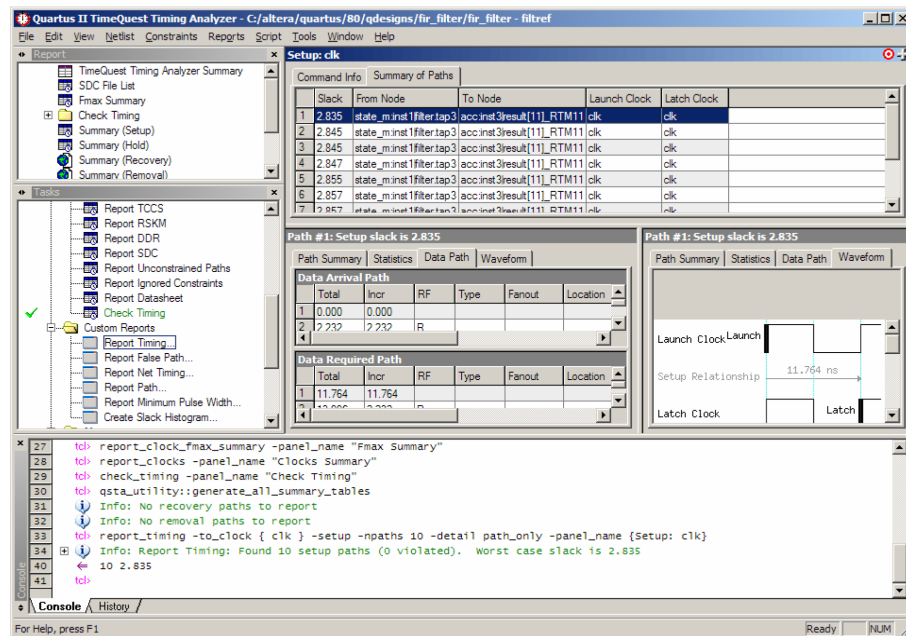
Each phase in the compilation flow can also be started independently, similar to the ISE software task flow windows. The Tasks window in the Quartus II software allows you to use the tools and features of the software and monitor progress from a flow-based layout.

Static Timing Analysis

Similar to the Post-Place and Route Static Timing Report generated by the Xilinx TRACE timing analyzer, the TimeQuest timing analyzer analyzes and reports the performance of all logic in your design, allowing you to determine all of the critical paths that limit your design’s performance.

The TimeQuest timing analyzer is an easy-to-use, second-generation, ASIC-strength static timing analyzer that supports the industry-standard Synopsys Design Constraints (SDC) format. Figure 12 shows the TimeQuest timing analyzer GUI, which consists of the **View** pane, **Tasks** pane, **Console** pane, and **Report** pane.

Figure 12. TimeQuest Timing Analyzer GUI



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

The TimeQuest timing analyzer generally calculates all possible register-to-register and complex clock structures using the most conservative assumptions by default. In contrast, the Xilinx TRACE timing analyzer does not analyze many of these complex structures extensively. Table 5 summarizes the major differences in timing analysis between the Xilinx TRACE timing analyzer and the Altera TimeQuest timing analyzer.

Table 5. Differences Between Xilinx Timing Analyzer and the TimeQuest Timing Analyzer (Part 1 of 2)

Timing Analysis	Xilinx Timing Analyzer (2)	TimeQuest Timing Analyzer (1)
Cross-Domain Clock Analysis	All clocks are not analyzed by default.	All clocks are related by default.
Combinational Loop Structures	Are not analyzed.	Analyzed by default.
Recovery and Removal Analysis	Disabled by default.	Enabled by default.
Multicorner Timing Analysis	Supports all of the device's operating conditions.	Supports all of the device's operating conditions if enabled.

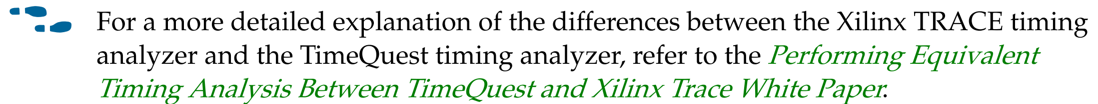
Table 5. Differences Between Xilinx Timing Analyzer and the TimeQuest Timing Analyzer (Part 2 of 2)

Timing Analysis	Xilinx Timing Analyzer (2)	TimeQuest Timing Analyzer (1)
Rise and Fall Analysis	Supported.	Supported.

Notes to Table 5:

(1) The TimeQuest timing analyzer supports multicorner timing analysis, which verifies that the timing constraints specified for a design meet a range of the device's operating conditions (for example, process, voltage, and temperature)

(2) The Xilinx timing analyzer supports only slow and fast corner; different operating condition timing corner is not supported.



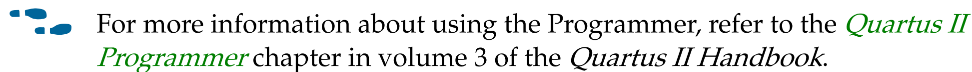
For a more detailed explanation of the differences between the Xilinx TRACE timing analyzer and the TimeQuest timing analyzer, refer to the *Performing Equivalent Timing Analysis Between TimeQuest and Xilinx Trace White Paper*.

Generation of Device Programming Files

Similar to `BitGen` and `PROMGen` in the Xilinx ISE software, the Quartus II Assembler generates programming files that the Quartus II Programmer can use to program or configure a device with Altera programming hardware.

The Assembler automatically converts the Fitter's device, logic cell, and pin assignments into a programming image for the device, in the form of one or more Programmer Object Files (**.pof**) or SRAM Object Files (**.sof**) for the target device. The **.sof** file is used to configure all Altera FPGA devices and the **.pof** file is used to program all Altera CPLD devices.

You can start a full compilation in the Quartus II software, which includes the Assembler module, or you can run the Assembler separately. To run the Assembler separately, on the Processing menu, point to **Start** and click **Start Assembler**.



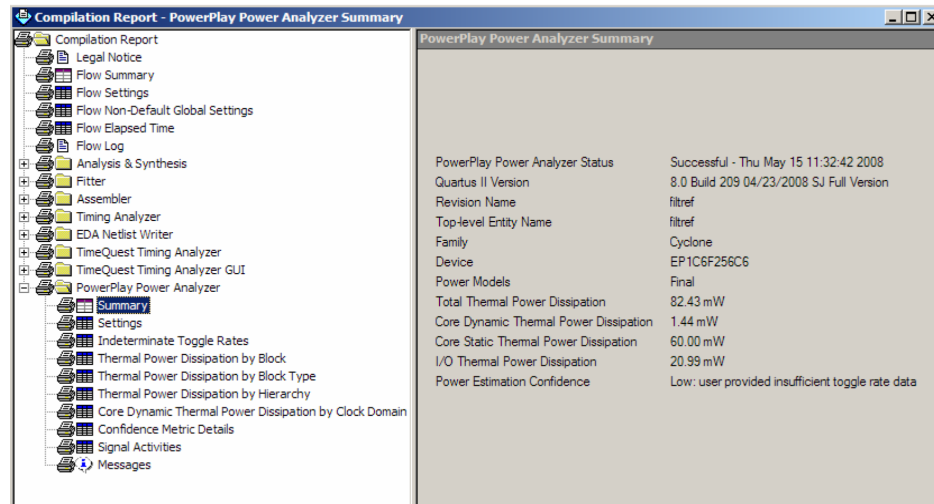
For more information about using the Programmer, refer to the *Quartus II Programmer* chapter in volume 3 of the *Quartus II Handbook*.

Power Analysis

Similar to the XPower Analyzer in the Xilinx ISE software, the Quartus II PowerPlay Power Analysis tools provide an interface that allows you to estimate static and dynamic power consumption throughout the design cycle.

The PowerPlay Power Analyzer performs post-fitting power analysis and produces a power report that highlights, by block type and entity, the power consumption of your design. Figure 13 shows the PowerPlay Power Analyzer Summary report.

Figure 13. PowerPlay Power Analyzer Summary Report



For more information about using the Quartus II PowerPlay Power Analysis tools, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

For more information about power driven synthesis and other power saving optimization techniques, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

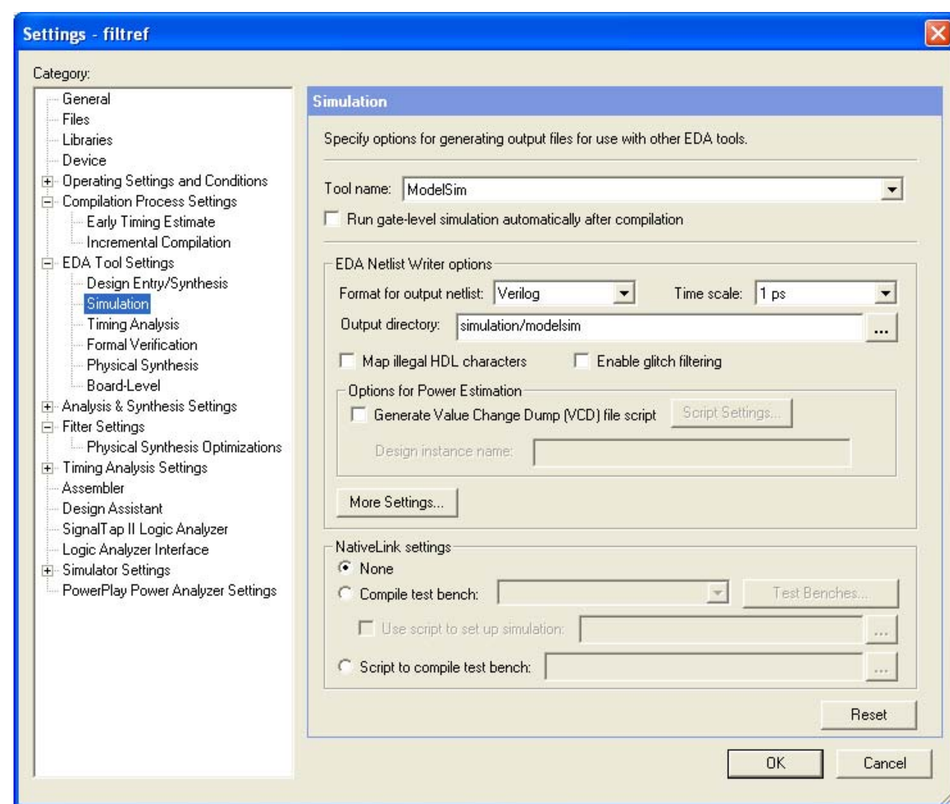
Simulation

As does the Xilinx ISE software, the Quartus II software supports integration with many third-party EDA simulation tools, such as Mentor Graphics® ModelSim, Cadence NC-Sim, and Synopsys VCS. In addition, the Quartus II software also supports the Aldec Active-HDL and Riviera-PRO simulation tools.

In the Quartus II software, you can specify the third-party simulation tools. To do this, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, expand the "+" icon for **EDA Tool Settings** and select **Simulation**. Figure 14 shows the **EDA Tool Settings Simulation** page. You can also specify this on the appropriate page of the New Project Wizard.

Figure 14. EDA Tool Settings Simulation Page



To perform functional/behavioral simulation on designs containing LPMs or MegaWizard-generated functions, use the Altera functional simulation models installed with the Quartus II software.

The LPM simulation model files are:

- **220model.v** for Verilog HDL
- **220pack.vhd** and **220model.vhd** for VHDL

The Altera megafunction simulation model files are:

- **altera_mf.v** for Verilog HDL

- **altera_mf.vhd** and **altera_mf_components.vhd** for VHDL

To perform gate-level timing simulation on a design, the Quartus II software generates output netlist files containing information about how the design was placed into device-specific architectural blocks.

The following are examples of generated output files:

- Verilog HDL output file (**.vo**)
- VHDL output file (**.vho**)
- Standard delay format output file (**.sdo**)

You can perform simulations with pre-compiled model libraries by using the Modelsim-Altera software included as part of the Quartus II software. You can also compile your own selection of model libraries with the Simulation Library Compiler tool in the Quartus II software.

For more information about using simulation tools and compiling simulation models, refer to the *Simulating Altera Designs* chapter, and simulation tool chapters in Volume 3 of the *Quartus II Handbook*.

Hardware Verification

Similar to the ChipScope Pro Tool in the Xilinx ISE software, the SignalTap® II Logic Analyzer is a multiple-input, digital acquisition instrument that captures and stores signal activity from any internal device node or nodes. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without using external equipment.

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

Viewing and Editing Design Placement

Similar to FloorPlanner and FPGA Editor in the Xilinx ISE software, the Quartus II software allows you to make small modifications, often referred to as engineering change orders (ECOs), to a design after a full compilation. These ECO changes are made directly to the design database, rather than to the source code or the Quartus II Settings File (**.qsf**). Making the ECO change to the design database allows you to avoid running a full compilation to implement the change.

To open the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan and Chip Editor)**, or you can locate a resource in the Chip Planner using the **Locate in Chip Planner (Floorplan and Chip Editor)** command on the Shortcut menu in many of the editors and windows in the Quartus II software.

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

Technique to Improve Productivity and Optimize Design

To improve productivity, the Quartus II incremental compilation feature reduces compilation time and run-time memory consumption by allowing you to recompile only the design partitions that have changed.

To optimize the design, the physical synthesis features and Design Space Explorer (DSE) tool allow you to optimize design performance. The following sections explain each feature.

Quartus II Incremental Compilation

In place of the SmartCompile feature in the ISE software, Quartus II Incremental Compilation allows you to split a large design into smaller partitions so that team members can work on the partitions independently. This simplifies the design process and reduces compilation time. The incremental compilation feature also preserves the results and performance for unchanged logic in your design as you make changes to other parts of the design. This allows you to perform more design iterations per day and achieve timing closure more efficiently. To take advantage of the compilation time savings and performance preservation, plan for an incremental flow early in your design cycle.

In addition, the design partition planner in the Quartus II software can help you to make informed decisions about how to partition your design. The design partition planner allows you to visualize a design's structure and create effective design partitions for use with incremental compilation.

Incremental compilation supports both top-down and bottom-up design flows. In a top-down design flow, a single designer compiles and optimizes the top-level design as a whole. A bottom-up design flow allows you to create a top-level design that instantiates any number of lower-level projects as design partitions. You can then design and optimize each partition as an independent Quartus II project and later integrate it into the top-level design with the Quartus II software export and import features, enabling team-based development.

In a bottom-up design flow, you must also assign fixed and locked LogicLock™ regions for each partition in the top-level design to avoid scattered or overlapping placement of partitions on the device by the Fitter when integrating partitions into the top-level design.

For more information about design planning and different design approaches, refer to the *Quartus II Incremental Compilation for Hierarchical and Team Based Design* chapters in volume 1 of the *Quartus II Handbook*.

Physical Synthesis Optimization

The Quartus II software offers advanced netlist optimization options, including physical synthesis, to optimize your design beyond the optimization performed in the course of the standard Quartus II compilation flow.

The synthesis netlist optimizations occur during the synthesis and Fitter stages of the Quartus II compilation flow. During the synthesis stage, the synthesis netlist optimizations make changes to the synthesis netlist output from a third-party synthesis tool or make changes as an intermediate step in Quartus II integrated synthesis. During the Fitter stage, optimizations make placement-specific changes to the netlist that improve performance results for your targeted Altera device.

To view and modify the synthesis netlist optimization options, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. In the **Category** list, expand the "+" icon for **Analysis & Synthesis Settings** and select **Synthesis Netlist Optimizations**.
3. On the **Synthesis Netlist Optimizations** page, specify the options for performing netlist optimization during synthesis.

For more information about netlist optimizations and physical synthesis, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Design Space Explorer (DSE)

In place of SmartXplorer, Design Space Explorer (DSE) is a utility written in Tcl/Tk that automatically finds the best set of Quartus II options for your design implementation. DSE explores the design space of your design by applying various optimization techniques and analyzing the results.

You can run DSE at any step in the design process; however, because the gains realized from optimizing Quartus II software settings might not persist over large changes in a design, Altera recommends that you run DSE late in the design cycle. Before running DSE, Altera recommends specifying the timing constraints for the design.

For more information, refer to the *Design Space Explorer* chapter in volume 3 of the *Quartus II Handbook*.

Additional Quartus II Features

In addition to providing the standard set of tools required in any FPGA design flow, the Quartus II software includes additional features and tools to assist you with achieving your desired design requirements.

Scripting with Tcl in the Quartus II Software

The Quartus II GUI provides an easy way to access all features and commands offered by the software. However, as designs grow in resource utilization and complexity, the need to automate common tasks and streamline the entire FPGA design flow becomes a requirement.

The Quartus II software provides support for Tcl to help facilitate project assignments, compilation, and constraints.

The Quartus II software contains Tcl application program interface (API) functions that you can use to automate a variety of common tasks, such as making assignments, compiling designs, analyzing timing, and controlling simulation. You can run your Tcl scripts in the following ways:

- Interactively from the shell
- Using scripts in batch from the shell
- As a batch file from the DOS or UNIX prompt
- Directly from the command line

For more information, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and the Quartus II Scripting Support page on the Altera website.

Running Tcl Scripts Interactively from the Shell

Using the -s or --shell switch option starts an interactive Tcl shell session, replacing the normal command line prompt with tcl>, as shown in the following example:

```
C:\>quartus_sh -s
Info: *******************************************************************
Info: The Quartus II Shell supports all TCL commands in addition
Info: to Quartus II Tcl commands. All unrecognized commands are
Info: assumed to be external and are run using Tcl's "exec"
Info: command.
Info: - Type "exit" to exit.
Info: - Type "help" to view a list of Quartus II Tcl packages.
Info: - Type "help <package name>" to view a list of Tcl commands
Info:   available for the specified Quartus II Tcl package.
Info: - Type "help -tcl" to get an overview on Quartus II Tcl usages.
Info: *******************************************************************

tcl>
```

Everything typed in the Tcl shell is directly interpreted by the Quartus II Tcl interpreter.

The Tcl shell includes a history list of previous commands entered, but it does not allow commands to span more than one line.

Using Scripts in Batch from a Shell

After you create a Tcl script file (**.tcl**), you can run it by typing the following command in a Tcl shell:

```
source <script_name>.tcl
```

Running Scripts from the DOS or UNIX Prompt

The following command runs the Quartus II Tcl shell and uses the Tcl file specified by the -t option as the input Tcl script:

```
quartus_sh -t <script_name>.tcl
```

The Quartus II Tcl interpreter reads and executes the Tcl commands in the Tcl script file and then exits back to the command-line prompt.

Running Scripts Directly from the Command Line

Another way to access Tcl is to use the --tcl_eval option. This directly evaluates the rest of the command line arguments as one or more Tcl commands. If there are two or more Tcl commands, separate them with semicolons.

For example, typing the following command:

```
quartus_sh --tcl_eval puts Hello; puts World
```

results in the following output:

```
Hello
World
```

The Tcl evaluate option allows external scripting programs (such as make, perl, and sh) to access information from the Quartus II software. These programs are used to obtain device family information for a targeted part. The --tcl_eval option also provides Tcl help information directly from the command-line prompt.

Using the Tcl Console in the Quartus II GUI

You can execute Tcl commands directly in the Quartus II Tcl Console window. To open the Tcl Console window, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Tcl Console is usually located in the bottom-right corner of the Quartus II GUI.

The Tcl script in Example 1 performs these tasks:

- Opens the fir_filter project, if it exists. If the project does not exist, the script creates the project.
- Sets the project to target a Stratix V 5SGXEA7K2F40C2 device
- Assigns the clk pin to the physical pin AV29
- Performs compilation

Example 1. Tcl Script Example

```
# This Tcl file works with quartus_sh.exe
# This Tcl file will compile the Quartus II tutorial
# fir_filter design

# set the project_name to fir_filter
# set compiler setting to filtref
set project_name fir_filter
set csf_name filtref

# Create a new project and open it
# Project_name is project name
# Require package ::quartus::project
if {![project_exists $project_name]} {
project_new -cmp $csf_name $project_name;
} else {
project_open -cmp $csf_name $project_name;
}
#------ Make device assignments ------#
set_global_assignment -name FAMILY "Stratix V"
set_global_assignment -name DEVICE 5SGXEA7K2F40C2

#------ Make instance assignments ------#
# assign pin clk to pin location AV29

set_location_assignment -to clk Pin_AV29

#------ project compilation ------#
# The project is compiled here
package require ::quartus::flow
execute_flow -compile
project_close
```

Cross-Probing in the Quartus II Software

Cross-probing is the ability to select design elements from one tool and locate them in another tool. All features and tools in the Quartus II software are highly integrated, resulting in a design environment that provides seamless cross-probing abilities.

For example, with the cross-probing ability in the Quartus II software, you can locate design elements from the RTL Viewer to the Assignment Editor. This eliminates the search time for node names and pin names when applying design constraints in the Assignment Editor. To locate the design elements, use the right mouse click button.

For more information about Quartus II software features and tools, refer to the *Quartus II Handbook* or *Quartus II Help*.

Xilinx to Altera Design Conversion

Successfully converting a Xilinx-targeted design for use in an Altera device is a three-step process:

1. Replace Xilinx primitives with Altera primitives, megafunctions, or constraints.
2. Replace Xilinx Core Generator modules with Altera megafunctions generated with the Quartus II MegaWizard Plug-In Manager.
3. Set timing and device constraints using the Quartus II software corresponding to those found in the Xilinx design you are converting.

These steps are described in the following sections.

Replacing Xilinx Primitives

Primitives are the basic building blocks of a Xilinx design. They perform various dedicated functions in the device, such as shift registers, and implementing specific I/O standards for the Xilinx device I/O pins. Primitives are easily identified, because their names are standardized.

This section describes methods of converting common Xilinx primitives, such as single-ended I/O buffers and shift registers, to the Altera equivalents. Table 6 lists commonly used Xilinx primitives and describes their equivalent Altera design element.

Table 6. Commonly Used Xilinx Device-Specific Primitives with Altera Equivalents (Part 1 of 2)

Xilinx Primitive	Description	Altera Equivalent	Conversion Method
BUF, 4, 8, 16	General Purpose Buffer	wire/signal Assignment	HDL
IBUF, 4, 8, 16	Single and Multiple Input Buffers		
OBUF, 4, 8, 16	Single and Multiple Output Buffers		

Table 6. Commonly Used Xilinx Device-Specific Primitives with Altera Equivalents (Part 2 of 2)

Xilinx Primitive	Description	Altera Equivalent	Conversion Method
`BUFG`	Global Clock Buffer	wire/signal and Global Signal Assignment	HDL and Assignment Editor
`IBUFG_`*<selectable I/O standard>* (1)	Input Global Buffer with selectable interface	wire/signal, I/O Standard, and Global Signal Assignment	
`OBUFG_`*<selectable I/O standard>* (1)	Output Global Buffer with selectable interface	—	
`IBUF_`*<selectable I/O standard>* (1)	Input buffer with selectable interface	wire/signal and I/O Standard Assignment (2)	
`IOBUF_`*<selectable I/O standard>* (1)	Bidirectional buffer with selectable interface	wire/signal and I/O Standard Assignment (2)	
`OBUF_`*<selectable I/O standard>* (1)	Output buffer with selectable interface	wire/signal and I/O Standard Assignment (2)	
`IBUFDS, OBUFDS`	Differential I/O Buffer	wire/signal and I/O Standard Assignment	HDL and Assignment Editor
`SRL16`	16-bit Shift Register	LPM_SHIFTREG with 16:1 MUX	HDL and MegaWizard Plug-In Manager

Notes to Table 6:

(1) The *<selectable I/O standard>* attributes are device-specific. For device-specific I/O standard information, refer to the Xilinx device's data sheet.

(2) For differential I/O buffer, you can assign differential I/O standard to the desired differential I/O signal. The Quartus II software automatically creates a new signal, *signal_name(n)*, that is opposite in phase with the desired signal.

I/O Buffer Conversion

Input, output, or bidirectional buffers are automatically inserted by the Quartus II compiler. Therefore, these buffers are not required when you convert your designs in the Quartus II software. Altera recommends removing all buffer primitives from the Xilinx design in your HDL code, and replace them with the simple wire/signal assignment.

For buffers with selectable I/O standards, Altera recommends replacing the buffers with wire assignments in your HDL code, and use the I/O Standard assignment in the Assignment Editor to assign the desired I/O standard.

For global buffers, Altera recommends replacing the global buffers with wire assignments in your HDL code, and use the Global Signal assignment in the Assignment Editor to assign the global signals. If the global buffer has a selectable I/O standard, use the I/O Standard assignment in the Assignment Editor to assign the desired I/O standard.

The following example converts `BUFG`, `IBUFG_SSTL2_II/I`, and `OBUF` in Verilog HDL.

Example 2 shows the original Verilog HDL code in the ISE software.

Example 2. I/O Buffer Conversion—Original Verilog HDL Code in the ISE Software

```
module Top (a, b, c, clk);
    input a, b, clk;
    output c;

    reg c_buf;
    wire a_buf, b_buf, clk_buf;

    BUFG inst1 (.O (clk_buf), .I (clk));
    IBUFG_SSTL2_II inst2 (.O (a_buf), .I (a));
    IBUFG_SSTL2_I inst3 (.O (b_buf), .I (b));
    OBUF inst4 (.O (c), .I (c_buf));

    always @ (posedge clk_buf)
        c_buf <= a_buf & b_buf;
endmodule
```

Example 3 shows the converted Verilog HDL code in the Quartus II software.

Example 3. I/O Buffer Conversion—Converted Verilog HDL Code in the Quartus II Software

```
module Top (a, b, c, clk);
    input a, b, clk;
    output c;

    reg c_buf;
    wire a_buf, b_buf, clk_buf;

    assign clk_buf = clk;
    assign a_buf = a;
    assign b_buf = b;
    assign c = c_buf;

    always @ (posedge clk_buf)
        c_buf <= a_buf & b_buf;
endmodule
```

The following example converts BUFG, IBUFG_SSTL2_II/I, and OBUF in VHDL.

Example 4 shows the original VHDL code in the ISE software.

Example 4. I/O Buffer Conversion—Original VHDL Code in the ISE Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Top IS
    PORT(
    a, b : IN STD_ULOGIC;
    clk : IN STD_ULOGIC;
    c : OUT STD_ULOGIC);
END Top;

ARCHITECTURE Behave OF Top IS

    SIGNAL a_buf, b_buf, c_buf, clk_buf : STD_ULOGIC;

    COMPONENT BUFG
        PORT (O : OUT STD_ULOGIC; I : IN STD_ULOGIC);
    END COMPONENT;

    COMPONENT IBUFG_SSTL2_II
        PORT (O : OUT STD_ULOGIC; I : IN STD_ULOGIC);
    END COMPONENT;

    COMPONENT IBUFG_SSTL2_I
        PORT (O : OUT STD_ULOGIC; I : IN STD_ULOGIC);
    END COMPONENT;

    COMPONENT OBUF
        PORT (O : OUT STD_ULOGIC; I : IN STD_ULOGIC);
    END COMPONENT;

    BEGIN
    inst1 : BUFG
       PORT MAP (O => clk_buf, I => clk);
    inst2 : IBUFG_SSTL2_II
       PORT MAP (O => a_buf, I => a);
    inst3 : IBUFG_SSTL2_I
       PORT MAP (O => b_buf, I => b);
    inst4 : OBUF
       PORT MAP (O => c, I => c_buf);

    PROCESS(clk_buf)
        BEGIN
        IF (clk_buf'event and clk_buf = '1') THEN
            c_buf <= a_buf AND b_buf;
        END IF;
    END PROCESS;
END Behave;
```

Example 5 shows the converted VHDL code in the Quartus II software.

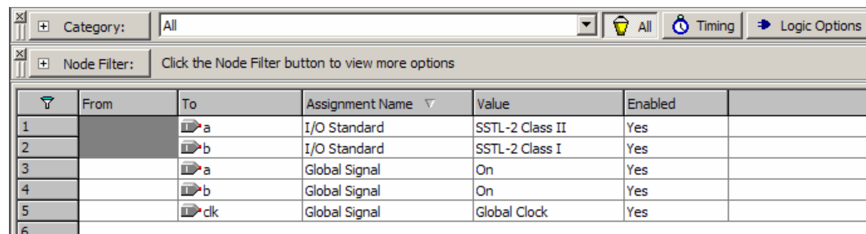
Example 5. I/O Buffer Conversion—Converted VHDL Code in the Quartus II Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Top IS
    PORT(
    a, b : IN STD_ULOGIC;
    clk : IN STD_ULOGIC;
    c : OUT STD_ULOGIC);
END Top;
ARCHITECTURE Behave OF Top IS
    SIGNAL a_buf, b_buf, c_buf, clk_buf : STD_ULOGIC;
    BEGIN
    PROCESS (a, b, c_buf, clk)
        BEGIN
        clk_buf <= clk;
        a_buf <= a;
        b_buf <= b;
        c <= c_buf;
    END PROCESS;

    PROCESS(clk_buf)
        BEGIN
        IF (clk_buf'event and clk_buf = '1') THEN
            c_buf <= a_buf AND b_buf;
        END IF;
    END PROCESS;
END Behave;
```

As you can see in the Verilog HDL or VHDL example, the `clk`, `a`, and `b` inputs are global signals, and the `a` and `b` inputs are also assigned with an I/O Standard `IBUFG_SSTL2_II` and `IBUFG_SSTL2_I`. To set the ports with specific assignments in the Quartus II software, use the Assignment Editor.

Figure 15 shows how to set the Global Signal and I/O Standard assignments using the Quartus II Assignment Editor. Inputs `a`, `b`, and `clk` are assigned as global signals, with `clk` as the global clock. Input ports `a` and `b` are assigned with specific I/O standards, while other ports are automatically assigned with the default device-specific I/O standard.

Figure 15. Global Signal and I/O Standard Assignments Using the Assignment Editor



The default I/O standard for pins on the target device in the Quartus II software is device-specific. To change the default I/O standard, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Device**. The **Device** page appears.

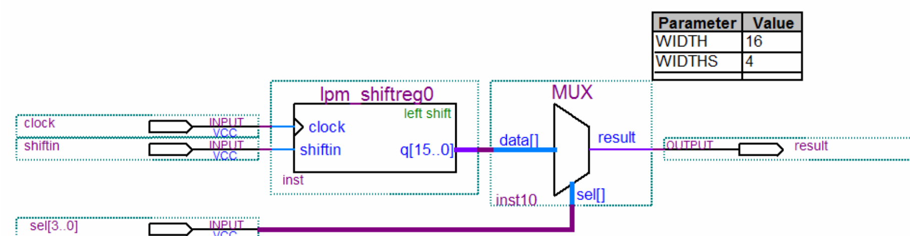
3. On the **Device** page, click **Device and Pin Options** and select the desired I/O standard on the **Voltage** tab.

SRL16 Conversion

To obtain the same functionality in the SRL16 primitive, use the LPM_SHIFTREG megafunction connected to a 16:1 multiplexer. The LPM_SHIFTREG megafunction is a parameterized shift register module in the Quartus II software. You can find this megafunction in the Storage folder of the MegaWizard Plug-In Manager.

Figure 16 is a high-level block diagram of the LPM_SHIFTREG megafunction connected to a 16:1 multiplexer.

Figure 16. LPM_SHIFTREG Megafunction Connected to a 16-Bit Multiplexer



In Figure 16, the LPM_SHIFTREG megafunction is configured as a left-shift register with a 16-bit parallel data output bus, a clock signal, and a 1-bit serial shift data input. The serial shift data input is shifted through the cascaded registers of the megafunction on every rising edge of the clock. The synchronous 16-bit parallel data output of the shift register is connected to a 16:1 multiplexer with a 4-bit selector. The multiplexer allows you to output the outcome of any selected register of the LPM_SHIFTREG megafunction by changing the selector value. As a result, the output of the multiplexer is the 1-bit serial shift output of the selected register.

You can also implement the 16:1 multiplexer with the LPM_MUX megafunction. For more information about using the LPM_MUX megafunction, refer to the Quartus II Help.

The following settings are required for the LPM_SHIFTREG megafunction to implement the block diagram in Figure 16:

- Set the size of the output bus, `q`, to **16**
- Set the direction of shifting to **left**
- Select **Data output** for output
- Select **Serial shift data input** for input

Table 7 summarizes the port mapping between the Xilinx SRL16 and the Altera equivalent using the LPM_SHIFTREG megafunction with a 16:1 multiplexer.

Table 7. Port Mapping Comparison

Xilinx Port	Altera Port	Description
Q	result	1-bit data output port
A0	sel[0]	4-bit data selector
A1	sel[1]	
A2	sel[2]	
A3	sel[3]	
CLK	clock	Clock signal for the shift register
D	shiftin	1-bit data input port
N/A	data[15..0] (1)	16-bit initial value of the shift register (optional)
N/A	load (1)	When asserted, the shift register initializes to the value of the data[15..0] port (optional)

Note to Table 7:

(1) The SRL16 primitive uses the INIT parameter in the HDL code to initialize the shift register. However, to initialize the LPM_SHIFTREG megafunction, Altera recommends adding the 16-bit parallel data input and load ports. When the load signal is asserted, the LPM_SHIFTREG megafunction is initialized with the value of the data[15..0] signal. These ports are optional, and if not included, LPM_SHIFTREG is initialized to zeroes by default.

For more information about using the LPM_SHIFTREG megafunction, refer to the *LPM_SHIFTREG Megafunction User Guide*.

The following example converts the SRL16 primitive to the LPM_SHIFTREG and LPM_MUX megafunctions in Verilog HDL.

Example 6 shows the original Verilog HDL code in the ISE software.

Example 6. SRL16 Conversion—Original Verilog HDL Code in the ISE Software

```
module top (
    A,
    CLK,
    D,
    Q
);

input[3:0] A;
input CLK;
input D;
output Q;

SRL16 i1 (.Q (Q),
          .A0 (A[0]),
          .A1 (A[1]),
          .A2 (A[2]),
          .A3 (A[3]),
          .CLK (CLK),
          .D (D));
endmodule
```

Example 7 shows the converted Verilog HDL code in the Quartus II software.

Example 7. SRL16 Conversion—Converted Verilog HDL Code in the Quartus II Software

```
module x2a_SRL16(
    A,
    CLK,
    D,
    Q
);

input[3:0] A;
input CLK;
input D;
output Q;

wire [15:0] shift_out;

lpm_shiftreg16i1(
    .clock(CLK),
    .shiftin(D),
    .q(shift_out));

lpm_mux16to1 i2(
    .data15(shift_out[15]),
    .data14(shift_out[14]),
    .data13(shift_out[13]),
    .data12(shift_out[12]),
    .data11(shift_out[11]),
    .data10(shift_out[10]),
    .data9(shift_out[9]),
    .data8(shift_out[8]),
    .data7(shift_out[7]),
    .data6(shift_out[6]),
    .data5(shift_out[5]),
    .data4(shift_out[4]),
    .data3(shift_out[3]),
    .data2(shift_out[2]),
    .data1(shift_out[1]),
    .data0(shift_out[0]),
    .sel(A),
    .result(Q));

endmodule
```

The following example converts the SRL16 primitive to the LPM_SHIFTREG and LPM_MUX megafunctions in VHDL.

Example 8 shows the original VHDL code in the ISE software.

Example 8. SRL16 Conversion—Original VHDL Code in the ISE Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY x2a_SRL16 IS
   port
   (
      A   :  IN  STD_LOGIC_VECTOR(3 downto 0);
      CLK :  IN  STD_LOGIC;
      D   :  IN  STD_LOGIC;
      Q   :  OUT  STD_LOGIC
   );
END x2a_SRL16;

ARCHITECTURE arch OF x2a_SRL16 IS

component SRL16
   PORT
   (
      A0  :  IN  STD_LOGIC;
      A1  :  IN  STD_LOGIC;
      A2  :  IN  STD_LOGIC;
      A3  :  IN  STD_LOGIC;
      CLK :  IN  STD_LOGIC;
      D   :  IN  STD_LOGIC;
      Q   :  OUT  STD_LOGIC
   );
end component;

BEGIN

i1 : SRL16
PORT MAP(A0 => A(0),
         A1 => A(1),
         A2 => A(2),
         A3 => A(3),
         CLK => CLK,
         D => D,
         Q => Q);

END;
```

Example 9 shows the converted VHDL code in the Quartus II software.

Example 9. SRL16 Conversion—Converted VHDL Code in the Quartus II Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY x2a_SRL16 IS
   port
   (
      A   :  IN  STD_LOGIC_VECTOR(3 downto 0);
      CLK :  IN  STD_LOGIC;
      D   :  IN  STD_LOGIC;
      Q   :  OUT  STD_LOGIC
   );
END x2a_SRL16;

ARCHITECTURE arch OF x2a_SRL16 IS

component lpm_shiftreg16
   PORT
   (
      clock : IN STD_LOGIC;
      shiftin : IN STD_LOGIC;
      q      : OUT STD_LOGIC_VECTOR(15 downto 0)
   );
end component;

component lpm_mux16to1
   PORT
   (
      data15 : IN STD_LOGIC;
      data14 : IN STD_LOGIC;
      data13 : IN STD_LOGIC;
      data12 : IN STD_LOGIC;
      data11 : IN STD_LOGIC;
      data10 : IN STD_LOGIC;
      data9 : IN STD_LOGIC;
      data8 : IN STD_LOGIC;
      data7 : IN STD_LOGIC;
      data6 : IN STD_LOGIC;
data5 : IN STD_LOGIC;
      data4 : IN STD_LOGIC;
      data3 : IN STD_LOGIC;
      data2 : IN STD_LOGIC;
      data1 : IN STD_LOGIC;
      data0 : IN STD_LOGIC;
      sel : IN STD_LOGIC_VECTOR(3 downto 0);
      result : OUT STD_LOGIC
   );
end component;

signal shift_out : STD_LOGIC_VECTOR(15 downto 0);
```

```
BEGIN

i1 : lpm_shiftreg16
PORT MAP(clock => CLK,
        shiftin => D,
        q => shift_out);


i2 : lpm_mux16to1
PORT MAP(data15 => shift_out(15),
        data14 => shift_out(14),
        data13 => shift_out(13),
        data12 => shift_out(12),
        data11 => shift_out(11),
        data10 => shift_out(10),
        data9 => shift_out(9),
        data8 => shift_out(8),
        data7 => shift_out(7),
        data6 => shift_out(6),
        data5 => shift_out(5),
        data4 => shift_out(4),
        data3 => shift_out(3),
        data2 => shift_out(2),
        data1 => shift_out(1),
        data0 => shift_out(0),
        sel => A,
        result => Q);
END;
```

Replace Xilinx Core Generator Modules

The following sections describe the conversion from Xilinx Core Generator modules to Altera megafunctions generated with the Quartus II MegaWizard Plug-In Manager:

- "Memory Conversion"
- "DCM to PLL Conversion" on page 53
- "Multiplier Conversion" on page 59

Memory Conversion

This section discusses the conversion of memory block from Xilinx to Altera. The targeted Xilinx memory blocks are only those generated through the Xilinx Block Memory Generator:

- Single-port RAM
- Dual-port RAM
- Single-port ROM
- Dual-port ROM

Altera memory blocks are categorized in the Memory Compiler folder in the Quartus II MegaWizard Plug-In Manager. The RAM and ROM blocks are comprised of the RAM:1-PORT, RAM:2-PORT, ROM:1-PORT, and ROM:2-PORT MegaWizard plug-ins.

For more information about the RAM or ROM megafunctions, refer to the *Random Access Memory Megafunction User Guide (RAM)* or the *Read Only Memory Megafunction User Guide (ROM)*.

Understanding the following topics helps you to successfully convert your Xilinx memory blocks to Altera memory blocks:

- Understanding the embedded memory blocks in your target device
- Understanding the differences between the memory features in Altera devices and Xilinx devices
- Creating Altera memory and identifying Xilinx-to-Altera port mapping

Features of Embedded Memory Blocks

Altera's embedded memory features different sizes of embedded memory blocks. Table 8 shows the embedded memory blocks in different Altera devices.

Table 8. Embedded Memory Blocks in Altera Devices

Devices	Types of TriMatrix Memory Blocks
Stratix, Stratix II, Cyclone®, Cyclone II, Arria® GX, and Arria II GX *(1)*	M512 blocks (512-bit) M4K blocks (4-Kbit) M-RAM blocks (512-Kbit)
Stratix III, Stratix IV, and Cyclone III *(2)*	MLAB blocks (640-bit) *(3)* M9K blocks (9-Kbit) M144K blocks (144-Kbit)
Arria V, Cyclone V, and Stratix V	MLAB blocks M10K blocks M20K blocks

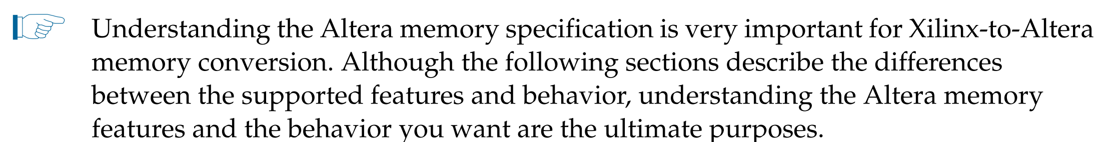
Notes to Table 8:

(1) Cyclone and Cyclone II devices have only M4K blocks.

(2) Cyclone III devices have only M9K blocks.

(3) For Stratix III, MLAB blocks are 640-bit in ROM mode and 320-bit in other modes.

For information about memory features and memory specification, refer to the appropriate *Embedded Memory Blocks* chapter for your device selection.



Understanding the Altera memory specification is very important for Xilinx-to-Altera memory conversion. Although the following sections describe the differences between the supported features and behavior, understanding the Altera memory features and the behavior you want are the ultimate purposes.

Refer to "Simple Dual-Port RAM Conversion" on page 47 for an example of simple dual-port RAM conversion.

The Quartus II software provides a feature that helps you verify the contents of the memories in the FPGA when you are in the test phase.

For more information about the memory content verifying feature, refer to the *In-System Updating of Memory and Constants* chapter in volume 3 of the *Quartus II Handbook*.

Differences Between Xilinx Memory and Altera Memory

There are differences in the features and behavior of Xilinx memory and Altera memory that you must take into consideration for the memory conversion. These differences include:

- Memory mode
- Clocking mode
- Write operation triggering
- Read-during-write operation at the same address
- Error Correction Code (ECC)
- Byte enable
- Address clock enable
- Parity bit support
- Memory initialization
- Output synchronous set/reset

The differences are detailed in the following sections.

Memory Mode

Xilinx memory and Altera memory support single-port RAM, simple dual-port RAM, true dual-port RAM, single-port ROM, and dual-port ROM.

For more information about the tri-port RAM, refer to the *Random Access Memory Megafunction User Guide (RAM)*.

Clocking Mode

Depending on which embedded memory block is selected, the following clock modes are available:

- Single clock
- Independent clock
- Input/output clock
- Read/write clock

In single clock mode, all ports share the common clock. In independent clock mode, a separate clock is available for each port (A and B). This is identical to Xilinx memory clock mode, in which each port (A and B) has an independent clock. Altera also supports input/output clock mode, in which a separate clock is available for input ports and output ports.

In read/write clock mode, a separate clock is available for read ports and write ports. Xilinx memory does not support a clock mode based on read/write operation. However, the clock mode is identical to Altera read/write clock mode if port A and port B are used as write port and read port, respectively, which is the case with simple dual-port RAM.

Write Operation Triggering

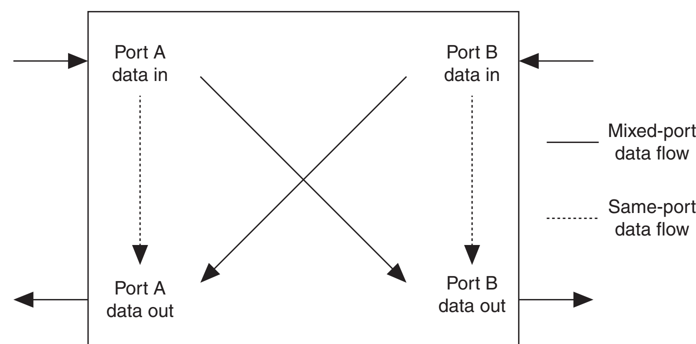
Potential write contentions must be resolved external to the RAM because writing to the same address location at both ports results in unknown data storage at that location. Therefore, knowing when the write operation was triggered is crucial.

Write operation in Altera memory can occur at either falling clock edges or rising clock edges, depending on the type of embedded memory block used.

Read-During-Write Operation at the Same Address

There are two types of read-during-write operations: same-port operations and mixed-port operations. Figure 17 shows the difference between these operations.

Figure 17. Read-During-Write Data Flow



The same-port read-during-write mode applies to either a single-port RAM or the same port of a true-dual port RAM. Mixed-port read-during-write mode applies to a RAM in simple or true-dual port mode that has one port reading and the other port writing to the same address location with the same clock.

Altera RAM and Xilinx RAM support both modes. However, they have different output options. The output options vary depending on the operation mode and type of embedded memory block or device selected.

Altera RAM is configured with output options of NEW_DATA (flow-through), OLD_DATA, or DONT_CARE. Xilinx RAM is configured with output options of READ_FIRST, WRITE_FIRST, or NO_CHANGE.

Table 9 lists the output options in Xilinx RAM and Altera RAM for a read-during-write operation.

Table 9. Output Options in Xilinx RAM and Altera RAM for Read-During-Write Operation (Part 1 of 2)

Description	Output Choices	
	Xilinx RAM	Altera RAM
The new data is written into memory and displayed at the output simultaneously	WRITE_FIRST	NEW_DATA
Outputs reflect the old data at that address before the new data is written into memory	READ_FIRST	OLD_DATA

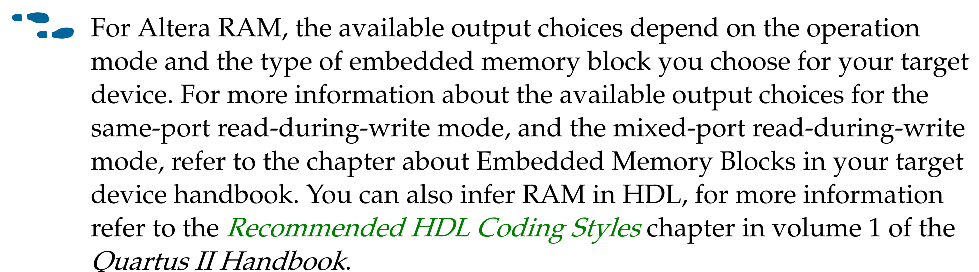
Table 9. Output Options in Xilinx RAM and Altera RAM for Read-During-Write Operation (Part 2 of 2)

Description	Output Choices	
	Xilinx RAM	Altera RAM
Outputs reflect the previous read data, and remain unaffected by the write operation	NO_CHANGE	Not supported (1)
The new data is written into memory, and the output displays unknown values	Not supported	DONT_CARE (2)

Notes to Table 9:

(1) To implement this behavior, add additional logic to retain its previous read data when read-during-write to the same address occurs. Use the write enable signal and the comparison of the write address and read address to track the operation when read-during-write to the same address occurs.

(2) You can choose DONT_CARE for a read-during-write operation if the output is not crucial to your design.

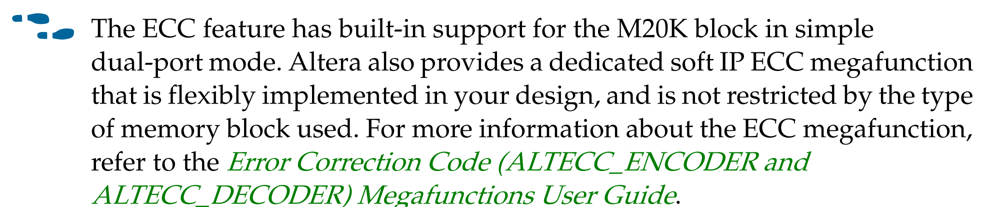


For Altera RAM, the available output choices depend on the operation mode and the type of embedded memory block you choose for your target device. For more information about the available output choices for the same-port read-during-write mode, and the mixed-port read-during-write mode, refer to the chapter about Embedded Memory Blocks in your target device handbook. You can also infer RAM in HDL, for more information refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Error Correction Code (ECC)

Altera RAM supports ECC only when the M20K type of Embedded Memory Block is selected in simple dual-port mode. Xilinx RAM supports ECC only for Virtex-5 devices in single-port RAM or simple dual-port RAM. ECC in Xilinx RAM and Altera RAM are generally meant to detect errors in the memory array and present the corrected single-bit error data on the output. However, the status signal used in Xilinx RAM and Altera RAM is different. Xilinx RAM uses two status outputs (`SBITERR` and `DBITERR`) to indicate the status of the data read. Altera M20K status is communicated via the three-bit status flag `eccstatus[2..0]`.

ECC cannot be used with the byte-enable feature. In addition, read-during-write “old data” mode is not supported if ECC is selected.



The ECC feature has built-in support for the M20K block in simple dual-port mode. Altera also provides a dedicated soft IP ECC megafunction that is flexibly implemented in your design, and is not restricted by the type of memory block used. For more information about the ECC megafunction, refer to the *Error Correction Code (ALTECC_ENCODER and ALTECC_DECODER) Megafunctions User Guide*.

Table 10 shows the truth table for the ECC status flags.

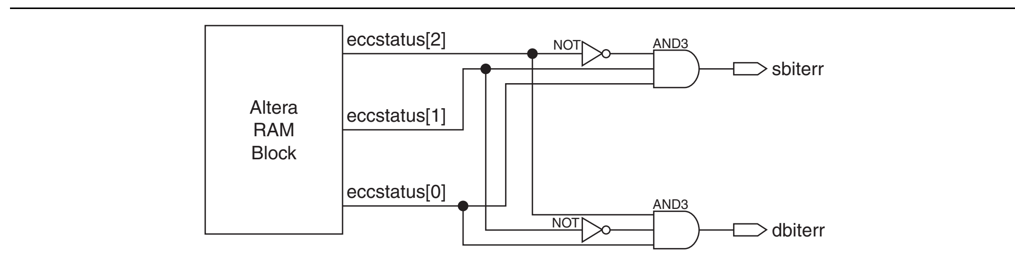
Table 10. Truth Table for ECC Status Flags

Status	eccstatus[2]	eccstatus[1]	eccstatus[0]
No error	0	0	0
Single error and fixed	0	1	1
Double error and no fix	1	0	1
Illegal	0	0	1
Illegal	0	1	0
Illegal	1	0	0
Illegal	1	1	X

Because Xilinx RAM uses two status outputs (SBITERR and DBITERR) to indicate the status of the data read, and Altera M20K status is communicated via a three-bit status flag eccstatus[2..0], additional logic is required for the eccstatus[2..0] ports before port mapping to the sbiterr and dbiterr ports.

Figure 18 shows suggested additional logic to complete the port mapping from eccstatus[2..0] ports to your current sbiterr and dbiterr ports. When an illegal status of eccstatus [2..0] occurs, the sbiterr and dbiterr signals are de-asserted.

Figure 18. Suggested Xilinx-to-Altera Port-Mapping for the ECC Status Signal



For more information about ECC, refer to the chapter about Embedded Memory Blocks in your target device handbook

Byte Enable

All embedded memory blocks support byte enables that mask the input data so that only specific bytes of data are written. The unwritten bytes or bits retain the previous written value. Xilinx RAM supports byte enables only in Virtex-4 and newer devices. There are differences in the byte enables supported in Xilinx RAM and Altera RAM.

Table 11 shows the general differences for the byte enables supported in Xilinx RAM and Altera RAM.

Table 11. Byte Enables Features in Xilinx RAM and Altera RAM

Differences	Xilinx RAM	Altera RAM
Controlling signals	Controlled by `WEA[n:0]` signal. Each bit in the `WEA[n:0]` acts as a write enable for the corresponding input data byte.	Controlled by write enable signal and byte-enable signal. (1)
Supported input data width	Support multiples of 8 or 9 bits.	Support multiples of 5 bits, 8 bits, and 9 bits excluding the 5 bits, 8 bits, and 9 bits. For a configuration less than two bytes wide, the write enable signal or clock enable signal is used to control the write operation. (2)
Controllable output values for the output byte correspond to the input masked byte when read-during-write to the same location occurs.	Not supported.	When a certain input byte is masked out, the corresponding data byte output can appear as either an unknown value (DONT_CARE) or the current data (OLD_DATA) at that location. (3)

Notes to Table 11:

(1) Asserts the write enable signal and the specific bit of the byte enable signal to control which byte is to be written. For example, if you are using a RAM block in x18mode, with the write enable asserted and the byte enable signal = 01, `data[8..0]` is enabled and `data[17..9]` is disabled.

(2) Byte enable for multiple 5 bits of input data width is supported only when the MLAB memory block type is selected.

(3) The output value for the masked byte is controllable via the MegaWizard Plug-In Manager, and is dependent on the selected memory block type.

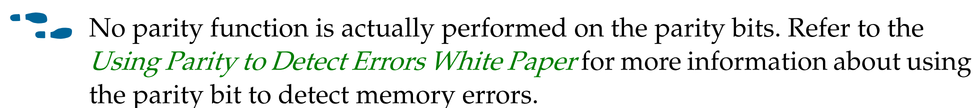
Address Clock Enable

Altera memory supports the address clock enable feature. The address clock enable holds the previous address value for as long as `addressstall` is enabled.

For more information about the address clock enable feature, refer to the chapter about Embedded Memory Blocks in your target device handbook.

Parity Bit Support

All embedded memory blocks have built-in parity-bit support for each byte. The parity bits are added to the amount of memory in each RAM block. In addition, for error detection to ensure data integrity, the parity bits can also be used for other purposes, such as storing user-specified control bits.



No parity function is actually performed on the parity bits. Refer to the *Using Parity to Detect Errors White Paper* for more information about using the parity bit to detect memory errors.

Memory Initialization

All embedded memory blocks support memory initialization. The memory contents is initialized using a memory initialization file (**.mif**) or Hexadecimal (Intel-Format) file (**.hex**) created in the Quartus II software. Specify the file name while configuring your memory megafunction through the MegaWizard Plug-In Manager. Xilinx memory content is initialized using a memory coefficient (COE) file or by using the default data option.

Output Synchronous Set/Reset

Xilinx Memory supports optional synchronous set/reset pins that control the reset operation of the last register in the output stage. This initializes the output of the memory to a user-defined value when the SSR signal is asserted. Altera memory does not have this feature. However, as a workaround to mimic the functionality of the SSR signal, you can add additional logic, such as using a 2-to-1 multiplexer, to select the unregistered output of the memory or the user-defined value with the SSR signal connected to the multiplexer selection control signal. You might also want to use the additional logic for the enable signals (REGCEA, REGCEB, or EN). In addition, you must register the outputs to provide the equivalent clock latency in the Xilinx RAM.

Memory Creation and Port-Mapping

Understanding the memory features of the embedded memory blocks in your target device is crucial for Xilinx-to-Altera memory conversion. This helps you to identify the type of memory block to use in replacing your current Xilinx memory block.

If you are not sure which memory block to select, or are not particular about the memory block type, you can select **AUTO** in the MegaWizard Plug-In Manager. With this option, the memory block type is determined by the Quartus II software synthesizer or Fitter at compile time. Check the Quartus II Fitter Report to determine the type of memory block that was assigned to your design.

You can build the memory blocks in the MegaWizard Plug-In Manager using the proper MegaWizard plug-in, as shown in Table 12.

Table 12. MegaWizard Plug-Ins Used For Respective Memory Modes/Functions

Memory Modes/Functions	MegaWizard Plug-In Used
Single-port RAM	RAM:1-PORT
Simple dual-port RAM	RAM:2-PORT
True dual-port RAM	RAM:2-PORT
Single-port ROM	ROM:1-PORT
Dual-port ROM	ROM:2-PORT

For more information about memory options, and how to build the memory function through the MegaWizard Plug-In Manager, refer to the *Random Access Memory (RAM) Megafunction User Guide* and the *Read Only Memory (ROM) Megafunction User Guide*.

After you have created the memory function, identify the port-mapping from Xilinx memory ports to Altera memory ports.

Table 13 lists the memory ports generated through the Block Memory Generator and their corresponding mapping to Altera memory ports for different memory modes.

Table 13. Block Memory Generator's Memory Port Mapping to Altera Memory Ports (1) (Part 1 of 2)

Port Description	Xilinx Ports	Port-Mapping to Altera Ports In Different Memory Modes				
		Single-Port RAM	Simple Dual-Port RAM	True Dual-Port RAM	Single-Port ROM	Dual-Port ROM
Port A: address	addra	address	wraddress	address_a	address	address_a
Port A: data input	dina	data	data	data_a	NA (2)	NA
Port A: clock enable	ena	clken	wrclocken	enable_a	clken	enable_a
Port A: clock enable for the last output register	regcea (1)	—	NA	—	—	—
Port A: write enable	wea (3)	wren and/or byteena	wren and/or byteena_a	wren_a and/or byteena_a	NA	NA
Port A: synchronous set/reset	ssra	—	NA	—	—	—
Port A: clock	clka	clock	wrclock	clock_a	clock	clock_a
Port A: data output	douta	q	NA	q_a	q	q_a
Port B: address	addrb	NA	rdaddress	address_b	NA	address_b
Port B: data input	dinb	NA	NA	data_b	NA	NA
Port B: clock enable	enb	NA	rdclocken	enable_b	NA	enable_b
Port B: clock enable for the last output register	regceb (1)	NA	—	—	NA	—
Port B: write enable	web (3)	NA	NA	wren_b and/or byteena_b	NA	NA
Port B: synchronous set/reset	ssrb	NA	—	—	NA	—
Port B: clock	clkb	NA	rdclock	clock_b	NA	clock_b
Port B: data output	doutb	NA	q	q_b	NA	q_b

Table 13. Block Memory Generator's Memory Port Mapping to Altera Memory Ports (1) (Part 2 of 2)

Port Description	Xilinx Ports	Port-Mapping to Altera Ports In Different Memory Modes				
		Single-Port RAM	Simple Dual-Port RAM	True Dual-Port RAM	Single-Port ROM	Dual-Port ROM
Single bit error	`sbiterr`	—	`eccstatus[2:0]` (4)	NA	NA	NA
Double bit error	`dbiterr`	—		NA	NA	NA

***Port mappings denoted with NA are not applicable for that memory mode; port-mappings denoted with — are not supported in Altera memory.**

Notes to Table 13:

(1) Altera memory does not support additional pipeline at the output port, and therefore, does not have clock enable for the last output register. However, Altera memory uses only the single clock enable pin to flexibly control the port A and port B registers. For certain configurations, you can select which input/output register of port A and port B to take effect on the clock enable signal. The available options are shown when you configure your RAM or ROM through the MegaWizard Plug-In Manager.

(2) Port mappings denoted with NA are not applicable for that memory mode; port-mappings denoted with — are not supported in Altera memory.

(3) For configurations less that two bytes wide, Xilinx write enable signals (`wea` and `web`) are equivalent to Altera write enable signals (`wren`, `wren_a`, or `wren_b`) signals, depending on the memory mode used. For configurations of more than two bytes, Xilinx's write enable buses (`wea[]` and `web[]`) are equivalent to Altera byte enable buses (`byteena[]`, `byteena_a[]`, or `byteena_b[]`), depending on the memory mode used. Also, the Altera write enable signal need to be asserted for the write operation.

(4) ECC status in embedded Altera memory is communicated via the three-bit status flag `eccstatus[2..0]`. Designs containing Xilinx RAM with the ECC feature must be modified for the Altera 3-bit ECC status signal. Refer to the section "Error Correction Code (ECC)" on page 42 for more information

Simple Dual-Port RAM Conversion

The following example has a top level, `test`, which instantiates `sdp_ram`, a Xilinx simple dual-port RAM (generated through Block Memory Generator). The example includes Verilog HDL and VHDL code for the top level that instantiates the Xilinx simple dual-port RAM.

The `sdp_ram` simple dual-port RAM is configured through the Block Memory Generator with the following properties:

- Simple dual-port RAM
- Input data width = 16 bits
- Memory depth = 8
- ECC feature selected
- Output registered (one stage pipeline)
- Read-during-write = read first (old data)

Example 10 shows the original Verilog HDL code in the ISE software.

Example 10. Simple Dual-Port RAM Conversion—Original Verilog HDL Code in the ISE Software

```
module test(clka,
             dina,
             addra,
             ena,
             wea,
             clkb,
             addrb,
             enb,
             doutb,
             dbiterr,
             sbiterr);

input clka;
   input [15 : 0] dina;
   input [2 : 0] addra;
   input ena;
   input [0 : 0] wea;
   input clkb;
   input [2 : 0] addrb;
   input enb;
   output [15 : 0] doutb;
   output dbiterr;
   output sbiterr;

   sdp_ram i1(
       .clka(clka),
       .dina(dina),
       .addra(addra),
       .ena(ena),
       .wea(wea),
       .clkb(clkb),
       .addrb(addrb),
       .enb(enb),
       .doutb(doutb),
       .dbiterr(dbiterr),
       .sbiterr(sbiterr));

endmodule
```

Example 11 shows the original VHDL code in the ISE software.

Example 11. Simple Dual-Port RAM Conversion—Original VHDL Code in the ISE Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY test IS
   port (
   clka: IN std_logic;
   dina: IN std_logic_VECTOR(15 downto 0);
   addra: IN std_logic_VECTOR(2 downto 0);
   ena: IN std_logic;
   wea: IN std_logic_VECTOR(0 downto 0);
   clkb: IN std_logic;
   addrb: IN std_logic_VECTOR(2 downto 0);
   enb: IN std_logic;
   doutb: OUT std_logic_VECTOR(15 downto 0);
   dbiterr: OUT std_logic;
   sbiterr: OUT std_logic);
END test;

ARCHITECTURE arch OF test IS

component sdp_ram
   PORT(

clka: IN std_logic;
          dina: IN std_logic_VECTOR(15 downto 0);
          addra: IN std_logic_VECTOR(2 downto 0);
          ena: IN std_logic;
          wea: IN std_logic;
          clkb: IN std_logic;
          addrb: IN std_logic_VECTOR(2 downto 0);
          enb: IN std_logic;
          doutb: OUT std_logic_VECTOR(15 downto 0);
          dbiterr: OUT std_logic;
          sbiterr: OUT std_logic
      );
end component;

BEGIN
i1 : sdp_ram
PORT MAP(clka => clka,
      dina => dina,
      addra => addra,
      ena => ena,
      wea => wea,
      clkb => clkb,
      addrb => addrb,
      enb => enb,
      doutb => doutb,
      dbiterr => dbiterr,
      sbiterr => sbiterr);
END;
```

To convert the simple dual-port RAM from Xilinx to Altera, you must create an Altera simple dual-port RAM through the Quartus II software MegaWizard Plug-In Manager. Configure the RAM from the wizard with the following options:

- With one read port and one write port

- Width of the output bus = 16 bits
- Number of 16-bit words = 8
- Memory block type = M20K
- Clocking method = Dual clock: use separate 'read' and 'write' clocks
- Enable ECC feature
- Output register
- Create one clock enable signal for each clock

For Altera RAM, read-during-write 'old data' mode is not supported if the ECC feature is used. Therefore, when you convert your Xilinx RAM to Altera RAM, you can expect to see a 'don't care' value when read-during-write to the same address occurs. If you really want the read-during-write 'old data' mode behavior, the workaround is to add additional logic to retain its previous read data when read-during-write to the same address occurs.

After you have created the simple dual-port RAM, you can replace the Xilinx RAM by instantiating the newly created Altera RAM, as shown in the following example.

Because Xilinx RAM uses two status outputs (`SBITERR` and `DBITERR`) to indicate the status of the data read, and Altera M20K status is communicated via the three-bit status flag `eccstatus[2..0]`, additional logic is required to carry the interfacing between the Altera RAM and your top level.

The following example shows the Verilog HDL and VHDL code.

Example 12 shows the converted Verilog HDL code in the Quartus II software.

Example 12. Simple Dual-Port RAM Conversion—Converted Verilog HDL Code in the Quartus II Software

```
module test(clka,
    dina,
    addra,
    ena,
    wea,
    clkb,
    addrb,
    enb,
    doutb,
    dbiterr,
    sbiterr);

    input clka;
    input [15 : 0] dina;
    input [2 : 0] addra;
    input ena;
    input [0 : 0] wea;
    input clkb;
```

```
output [15 : 0] doutb;
    output reg dbiterr;
    output reg sbiterr;

    wire [2:0] wire_eccstatus;

    always@(wire_eccstatus)
    begin
        if (wire_eccstatus==3'b000)
        begin
            dbiterr = 1'b0;
            sbiterr = 1'b0;
        end
        else if (wire_eccstatus==3'b011)
        begin
            dbiterr = 1'b0;
            sbiterr = 1'b1;
        end
        else if (wire_eccstatus==3'b101)
        begin
            dbiterr = 1'b1;
            sbiterr = 1'b0;
        end
        else
        begin
            dbiterr = 1'b0;
            sbiterr = 1'b0;
        end
    end

    sdp_ram i1(
        .wrclock(clka),
        .data(dina),
        .wraddress(addra),
        .wrclocken(ena),
        .wren(wea),
        .rdclock(clkb),
        .rdaddress(addrb),
        .rdclocken(enb),
        .q(doutb),
//replace sbiterr and dbiterr by eccstatus and
//connect it to wire_eccstatus
        .eccstatus(wire_eccstatus));

endmodule
```

Example 13 shows the converted VHDL code in the Quartus II software.

Example 13. Simple Dual-Port RAM Conversion—Converted VHDL Code in the Quartus II Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY test IS
   port (
   clka: IN std_logic;
   dina: IN std_logic_VECTOR(15 downto 0);
   addra: IN std_logic_VECTOR(2 downto 0);
   ena: IN std_logic;
   wea: IN std_logic;
   clkb: IN std_logic;
   addrb: IN std_logic_VECTOR(2 downto 0);
   enb: IN std_logic;
   doutb: OUT std_logic_VECTOR(15 downto 0);
   dbiterr: OUT std_logic;
   sbiterr: OUT std_logic);
END test;

ARCHITECTURE arch OF test IS

component sdp_ram
   PORT
   (
      data   : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
      rdaddress: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
      rdclock: IN STD_LOGIC ;
      rdclocken: IN STD_LOGIC  := '1';
      wraddress: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
      wrclock: IN STD_LOGIC ;
      wrclocken: IN STD_LOGIC  := '1';
      wren   : IN STD_LOGIC  := '1';

eccstatus: OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
      q          : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
   );
end component;

signal eccstatus_signal : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

   i1 : sdp_ram
   PORT MAP(wrclock => clka,
         data => dina,
         wraddress => addra,
         wrclocken => ena,
         wren => wea,
         rdclock => clkb,
         rdaddress => addrb,
```

```
            rdclocken => enb,
            q => doutb,
            eccstatus => eccstatus_signal);

    process(eccstatus_signal)
    begin
        if(eccstatus_signal = "000") then
            dbiterr <= '0';
            sbiterr <= '0';
        elsif(eccstatus_signal = "011") then
            dbiterr <= '0';
            sbiterr <= '1';
        elsif(eccstatus_signal = "101") then
            dbiterr <= '1';
            sbiterr <= '0';
        else
            dbiterr <= '0';
            sbiterr <= '0';
        end if;
    end process;

END;
```

DCM to PLL Conversion

To increase device and board-level performance, some Altera device families offer support for PLLs that allow you to minimize clock skew and clock delay and provide support for clock synthesis. Similar to Digital Clock Managers (DCM) in some Xilinx devices, you can easily convert DCMs to PLLs in Altera devices.

Table 14 compares the DCM features in Virtex-7 with the PLL features in Stratix V devices.

Table 14. DCM in Virtex-7 versus PLL in Stratix V Devices (Part 1 of 2)

Features		Xilinx DCM (Virtex-7)	Altera PLL (Stratix V)
Frequency Synthesis	Clock Multiplication and Division	✓	✓
	Phase and Delay Shifting	✓	✓
	Clock Duty Cycle	✓	✓
DCM Deskew Adjust	System Synchronous Normal Mode	✓	✓
	Source Synchronous	✓	✓
	Zero Delay Buffer	—	✓
	No Compensation	—	✓
	External Feedback	✓	✓

Table 14. DCM in Virtex-7 versus PLL in Stratix V Devices (Part 2 of 2)

Features		Xilinx DCM (Virtex-7)	Altera PLL (Stratix V)
Others	Input Clock Switchover	—	✓
	Dynamic re-configuration	✓	✓
	Single or Differential Clock Inputs	✓	✓
	Duty Cycle Correction	✓	—

In addition to DCM in Xilinx devices, Virtex-7 also has PLL components to support variable frequency synthesis, zero delay buffer, and to input jitter filtering. These features are also supported by PLLs in Altera devices, and you can easily convert from Xilinx to Altera.

For more information about specific PLL features in Altera devices, refer to the appropriate device handbook chapter.

Xilinx DCM to Altera PLL Port Mapping

You can easily convert DCMs that target a Xilinx device to PLLs in an Altera device by using the MegaWizard Plug-In Manager. Unlike the Xilinx DCM, which requires specific input buffers to feed into the source clock port of the DCM (for example, `IBUF`, `IBUFG`, or `BUFGMUX`), PLLs in Altera devices do not require input buffers when using the MegaWizard Plug-In Manager.

When converting DCMs, you can use the ALTPLL megafunction. This megafunction allows you to create and customize your PLL targeting to Altera devices.

Table 15 shows the port-mapping between DCM Virtex-7 and the PLL in Stratix V devices. The Xilinx ports are generated from the Xilinx Core Generator and the Altera ports are generated from the MegaWizard Plug-In Manager.

Table 15. Port-Mapping DCM Virtex-7 versus PLL Stratix V (Part 1 of 2)

Xilinx DCM Core Port	Altera ALTPLL Megafunction Port	Description
`CLKIN_IN`	`inclk0`	Clock Input to the DCM. This is equivalent to the Stratix V device's first clock input to the PLL.
N/A	`inclk1`	Second clock input to Stratix V device's PLL. This is not available in Virtex-7 DCM.
`CLKFB_IN`	`fbin`	External clock feedback.
N/A	`clkswitch`	Switch between input clock ports.
`RST_IN`	`areset`	Asynchronous reset port.
N/A	`pfdena`	Enables the phase frequency detector (PFD).

Table 15. Port-Mapping DCM Virtex-7 versus PLL Stratix V (Part 2 of 2)

Xilinx DCM Core Port	Altera ALTPLL Megafunction Port	Description
CLK0_OUT, CLK90_OUT, CLK180_OUT, CLK270_OUT, CLKDV_OUT, CLK2X_OUT, CLK2X180_OUT, CLKFX_OUT, CLKFX180_OUT	clk[n..0]	Clock frequency output ports. Xilinx DCM has fixed settings for most outputs and the ALTPLL megafunction is easily configured to suit them.
N/A	clkbad[1..0]	Specifies which clock input signal is not toggled.
LOCKED_OUT	locked	Provides the status of the PLL if it is locked.
N/A	activeclock	Specifies when the clock switch over circuit initiates.
CLK0_OUT	fbout	Specifies the output to the mimic circuitry and feeds into the feedback port.
N/A	phasecounterselect[3..0]	Specifies counter select.
PSINCDEC_IN	phaseupdown	Phase-shift increment and decrement.
PSCLK_IN	phasestep	Specifies dynamic phase shifting.
PSEN_IN	N/A	Phase-shift enable.
PSDONE_OUT	phasedone	Phase shift done output.
DCLK_IN, DI_IN[15:0], DEN_IN,DWE_IN, DO_OUT[15:0], DRDY_OUT, DADDR_IN[6:0]	Scanclk, scandata, scanclkena, configupdate, scandataout, scandone	Dynamic reconfiguration ports (1).

Note to Table 15:

(1) These ports are used for dynamic PLL reconfiguration. To perform the dynamic PLL reconfiguration in Altera devices, use the ALTPLL_RECONFIG megafunction. For more information about using dynamic PLL reconfiguration in Altera devices, refer to the *Phase-Locked Loops Reconfiguration Megafunction User Guide (ALTPLL_RECONFIG)*.

As shown in Table 15, all outputs of the ALTPLL megafunction is configured to support any of the clock synthesis ports of the DCM. This allows you to combine multiple DCMs into one instance of the ALTPLL megafunction. For example, to implement a multiplication factor of 2/3 and 6/5 would require two CLKFX ports in a Xilinx device. However, one ALTPLL instance can achieve the same functionality by applying the multiplication factor of 2/3 to clock c0 and applying 6/5 to clock c1, or vice versa.

In Virtex-7, the PLL component is also available for variable clock synthesis. The ALTPLL megafunction easily converts the PLL in Virtex-7, which is similar to the DCM conversion. In Altera designs, this can be done by a single PLL without cascading.

For more information about using the ALTPLL megafunction, refer to the *Phase-Locked Loops Megafunction User Guide (ALTPLL)*.

Xilinx DCM to Altera PLL Conversion

The following shows the original Verilog HDL and VHDL in the ISE software. The `mydcm` module is the module generated from the Xilinx Core Generator. The top module instantiates the `mydcm` module with `i1`. The input Clock Frequency is 100 MHz. The `CLKDV` is configured to divide by 2 (50 Mhz), and the `CLKFX` is configured to multiply by 4 (400 Mhz).

Example 14 shows the original Verilog code in the ISE software.

Example 14. Xilinx DCM to Altera PLL Conversion—Original Verilog Code in the ISE Software

```
module top( CLKIN_IN,
            RST_IN,
            CLK0_OUT,
            CLKDV_OUT,
            CLKFX_OUT,
            CLK2X_OUT,
            CLK90_OUT,
            LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
   output CLK0_OUT;
   output CLKDV_OUT;
   output CLKFX_OUT;
   output CLK2X_OUT;
   output CLK90_OUT;
   output LOCKED_OUT;

mydcm i1  (
    .CLKIN_IN(CLKIN_IN),
    .RST_IN(RST_IN),
    .CLK0_OUT(CLK0_OUT),
    .CLKDV_OUT(CLKDV_OUT),
    .CLKFX_OUT(CLKFX_OUT),
    .CLK2X_OUT(CLK2X_OUT),
    .CLK90_OUT(CLK90_OUT),
    .LOCKED_OUT(LOCKED_OUT)
    );

endmodule
```

Example 15 shows the original VHDL code in the ISE software.

Example 15. Xilinx DCM to Altera PLL Conversion—Original VHDL Code in the ISE Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY top IS
   port
   (
      CLKIN_IN    : IN  STD_LOGIC;
      RST_IN      : IN  STD_LOGIC;
      CLKDV_OUT   : OUT  STD_LOGIC;
      CLKFX_OUT   : OUT STD_LOGIC;
      CLK0_OUT    : OUT STD_LOGIC;
      CLK2X_OUT   : OUT STD_LOGIC;
      CLK90_OUT   : OUT STD_LOGIC;
      LOCKED_OUT : OUT  STD_LOGIC
   );
END top;

ARCHITECTURE arch OF top IS

component mydcm
   port
   (
      CLKIN_IN    : IN  STD_LOGIC;
      RST_IN      : IN  STD_LOGIC;
      CLKDV_OUT   : OUT  STD_LOGIC;
      CLKFX_OUT   : OUT STD_LOGIC;
      CLK0_OUT    : OUT STD_LOGIC;
      CLK2X_OUT   : OUT STD_LOGIC;
      CLK90_OUT   : OUT STD_LOGIC;
      LOCKED_OUT : OUT  STD_LOGIC
   );
end component;

BEGIN
i1 : mydcm
PORT MAP( CLKIN_IN => CLKIN_IN,
       RST_IN => RST_IN,
       CLKDV_OUT => CLKDV_OUT,
       CLKFX_OUT => CLKFX_OUT,
       CLK0_OUT => CLK0_OUT,
       CLK2X_OUT => CLK2X_OUT,
       CLK90_OUT => CLK90_OUT,
       LOCKED_OUT => LOCKED_OUT);
END;
```

You can also generate similar behavior in the ALTPLL megafunction using the MegaWizard Plug-In Manager. The following examples show the Verilog HDL and VHDL code in the Quartus II software. The `mypll` module is generated from the MegaWizard Plug-In Manager. The top module instantiates the `mypll` module with `i1`. The Clock Frequency input is 100 MHz and the clock output frequency synthesis is configured through `C0`, `C1`, `C2`, `C3`, and `C4` in the MegaWizard Plug-In Manager.

Example 16 shows the converted Verilog HDL code in the Quartus II software.

Example 16. Xilinx DCM to Altera PLL Conversion—Converted Verilog HDL Code in the Quartus II Software

```
module top( CLKIN_IN,
            RST_IN,
            CLK0_OUT,
            CLKDV_OUT,
            CLKFX_OUT,
            CLK2X_OUT,
            CLK90_OUT,
            LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
   output CLK0_OUT;
   output CLKDV_OUT;
   output CLKFX_OUT;
   output CLK2X_OUT;
   output CLK90_OUT;
   output LOCKED_OUT;

mypll i1  (
    .inclk0(CLKIN_IN),
    .areset(RST_IN),
     .c0(CLK0_OUT),
    .c1(CLKDV_OUT),
    .c2(CLKFX_OUT),

.c3(CLK2X_OUT),
    .c4(CLK90_OUT),
    .locked(LOCKED_OUT)
    );

endmodule
```

Example 17 shows the converted VHDL code in the Quartus II software.

Example 17. Xilinx DCM to Altera PLL Conversion—Converted VHDL Code in the Quartus II Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY top IS
   port
   (
      CLKIN_IN   : IN  STD_LOGIC;
      RST_IN     : IN  STD_LOGIC;
      CLKDV_OUT  : OUT  STD_LOGIC;
      CLKFX_OUT  : OUT STD_LOGIC;
      CLK0_OUT   : OUT STD_LOGIC;
      CLK2X_OUT  : OUT STD_LOGIC;
      CLK90_OUT  : OUT STD_LOGIC;
      LOCKED_OUT : OUT  STD_LOGIC
   );
END top;

ARCHITECTURE arch OF top IS

component mypll
   port
   (
      inclk0 : IN  STD_LOGIC;
      areset : IN  STD_LOGIC;
      c0     : OUT  STD_LOGIC;
      c1      : OUT STD_LOGIC;
      c2     : OUT STD_LOGIC;
      c3     : OUT STD_LOGIC;
      c4   : OUT STD_LOGIC;
      locked : OUT  STD_LOGIC
   );
end component;

BEGIN
i1 : mypll
PORT MAP( inclk0 => CLKIN_IN,
          areset => RST_IN,
          c0 => CLKDV_OUT,
          c1 => CLKFX_OUT,
          c2 => CLK0_OUT,
          c3 => CLK2X_OUT,
          c4 => CLK90_OUT,
          locked => LOCKED_OUT);
END;
```

Multiplier Conversion

The basic building blocks of all DSP applications are high-performance multiply-adders and multiply-accumulators. To address this requirement in FPGA devices, Altera devices offer dedicated DSP blocks, combining five arithmetic operations—multiplication, addition, subtraction, accumulation, and summation—into a single block.

Altera provides three Quartus II megafunctions for implementing various multiply, multiply-accumulate, and multiply-add functions using DSP blocks or logic resources:

- The LPM_MULT megafunction performs multiply functions only.
- The ALTMULT_ADD megafunction performs multiply or multiply-add functions.
- The ALTMULT_ACCUM megafunction performs multiply-accumulate functions

The following section discusses the conversion from Xilinx Multiplier Core to the Altera LPM_MULT megafunction.

For more information about specific DSP features, refer to the appropriate device handbook chapter. You can also infer DSP functions in HDL, for more information refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Xilinx Multiplier Core versus Altera LPM_MULT Megafunction

You can easily convert the Xilinx Multiplier Core that targets a Xilinx device into multipliers for an Altera device by using the Quartus II MegaWizard Plug-In Manager. Similar to the Xilinx Multiplier Core Generator, the Quartus II MegaWizard Plug-In Manager multipliers can use either logic elements or the dedicated multiplier blocks in the device.

When converting Xilinx Multiplier Core, you can use the LPM_MULT megafunction. With this megafunction, you can create multipliers that use logic elements or dedicated multipliers using Altera devices. Table 16 shows the comparison between the Xilinx Multiplier Core and the Altera LPM_MULT Megafunction.

Table 16. Xilinx Multiplier Core versus Altera LPM_MULT Megafunction

Feature	Xilinx Multiplier Core Generator Module	Altera LPM_MULT Megafunction
Constant Coefficient	✓	✓
Signed and Unsigned Data	✓	✓
Configurable Pipeline Latency	✓	✓
Area versus Speed Trade-off	✓	✓
Asynchronous Clear	— (1)	✓
Synchronous Clear	✓	— (1)
Part A and Port B support different sign	✓	— (2)

Notes to Table 16:

(1) The Xilinx Multiplier Core only supports synchronous clear, whereas the Altera LPM_MULT megafunction only supports asynchronous clear. If synchronous clear must be used, you can register the asynchronous clear signal before connecting the signal to the asynchronous clear port of LPM_MULT.

(2) The `dataa` and `datab` ports in the LPM_MULT megafunction must be on the same sign. If your design does not meet this requirement, you can consider using the ALTMULT_ADD megafunction to replace the Xilinx Multiplier Core.

Implementation and Port Mapping

You can only replace the Xilinx Multiplier Core with the LPM_MULT megafunction if the input ports are the same sign. Table 17 shows the port mapping between the Xilinx Multiplier Core and the Altera LPM_MULT megafunction.

Table 17. Port Mapping Between Xilinx Multiplier Core and LPM_MULT Megafunction

Xilinx Multiplier Core Port	Altera LPM_MULT Megafunction Port	Description
`A []`	`dataa []`	Data Input Port A
`B []`	`datab []`	Data Input Port B
`CLK []`	`clock`	Clock Port
`CE`	`clken`	Clock Enable Port
`SCLR`	`N/A`	Synchronous Clear Port
`N/A`	`aclr`	Asynchronous Clear Port
`P []`	`result []`	Multiplication Result Port

For more information about using the LPM_MULT megafunction, refer to the *LPM_MULT Megafunction User Guide*.

Xilinx Multiplier Core to ALTMULT Megafunction Conversion

The following shows the original Verilog HDL and VHDL code in the ISE software. The `mymult` module is generated from the Xilinx Core Generator. The `test` module instantiates the `mymult` module with `i1`.

Example 18 shows the original Verilog HDL code in the ISE software

Example 18. Xilinx Multiplier Core to ALTMULT Megafunction Conversion—Original Verilog HDL Code in the ISE Software

```
module test(
    input clk,
    input [17:0] a,
    input [17:0] b,
    input ce,
    input sclr,
    output [35:0] p
    );

mymult i1 (
    .clk(clk),
    .a(a), // Bus [17 : 0]
    .b(b), // Bus [17 : 0]
    .ce(ce),
    .sclr(sclr),
    .p(p)); // Bus [35 : 0]
endmodule
```

Example 19 shows the original VHDL code in the ISE software.

Example 19. Xilinx Multiplier Core to ALTMULT Megafunction Conversion—Original VHDL Code in the ISE Software

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY test IS
   port
   (
      clk :  IN  STD_LOGIC;
      a      :  IN  STD_LOGIC_VECTOR(17 downto 0);
      b      :  IN  STD_LOGIC_VECTOR(17 downto 0);
      ce     :  IN  STD_LOGIC;
      sclr:  IN  STD_LOGIC;
      p      :  OUT  STD_LOGIC_VECTOR(35 downto 0)
   );
END test;

ARCHITECTURE arch OF test IS

component mymult
   PORT(clk: IN STD_LOGIC;
        a  : IN STD_LOGIC_VECTOR(17 downto 0);
        b  : IN STD_LOGIC_VECTOR(17 downto 0);
        ce : IN STD_LOGIC;
        sclr : IN STD_LOGIC;
        p  : OUT STD_LOGIC_VECTOR(35 downto 0)
   );
end component;

BEGIN
i1 : mymult
PORT MAP(clk => clk,
        a => a,
        b => b,
        ce => ce,
        sclr => sclr,
        p => p);
END;
```

To convert Verilog HDL or VHDL code to be used in the Quartus II software, you must create the equivalent `mymult` module using the Altera LPM_MULT megafunction. The following example shows the Verilog HDL and VHDL code example that is compiled in the Quartus II software after the conversion.

Example 20 shows the converted Verilog HDL code in the Quartus II software (ALTMULT).

Example 20. Xilinx Multiplier Core to ALTMULT Megafunction Conversion—Converted Verilog HDL Code in the Quartus II Software

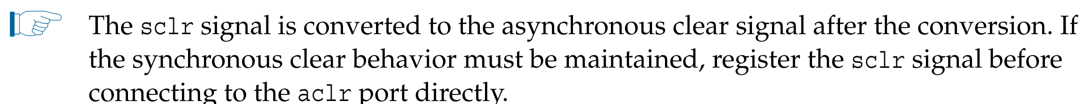
```
module test(
    input clk,
    input [17:0] a,
    input [17:0] b,
    input ce,
    input sclr,
    output [35:0] p
    );

mymult i1 (
    .clock(clk),
    .dataa(a), // Bus [17 : 0]
    .datab(b), // Bus [17 : 0]
    .clken(ce),
    .aclr(sclr),
    .result(p)); // Bus [35 : 0]
endmodule
```

Example 21 shows the converted VHDL code in the Quartus II software (ALTMULT).

Example 21. Xilinx Multiplier Core to ALTMULT Megafunction Conversion—Converted VHDL Code in the Quartus II Software

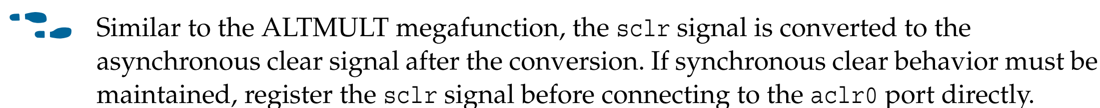
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY test IS
    port
    (
        clk :  IN  STD_LOGIC;
        a :  IN  STD_LOGIC_VECTOR(17 downto 0);
        b :  IN  STD_LOGIC_VECTOR(17 downto 0);
        ce :  IN  STD_LOGIC;
        sclr :  IN  STD_LOGIC;
        p :  OUT  STD_LOGIC_VECTOR(35 downto 0)
    );
END test;

ARCHITECTURE arch OF test IS

component mymult
    PORT(clock : IN STD_LOGIC;
         dataa : IN STD_LOGIC_VECTOR(17 downto 0);
         datab : IN STD_LOGIC_VECTOR(17 downto 0);
         clken : IN STD_LOGIC;
         aclr : IN STD_LOGIC;
         result : OUT STD_LOGIC_VECTOR(35 downto 0)
    );
end component;
```

```
BEGIN
i1 : mymult
PORT MAP(clock => clk,
        dataa => a,
        datab => b,
        clken => ce,
        aclr => sclr,
        result => p);
END;
```



The `sclr` signal is converted to the asynchronous clear signal after the conversion. If the synchronous clear behavior must be maintained, register the `sclr` signal before connecting to the `aclr` port directly.

As mentioned in the previous section, if the inputs' signs are different, you can use the Altera ALTMULT_ADD megafunction instead. The following example shows the Verilog HDL and VHDL example that is compiled in the Quartus II software after the conversion. The `mymult_add` module is created by the MegaWizard Plug-In Manager to implement the ALTMULT_ADD megafunction.



Similar to the ALTMULT megafunction, the `sclr` signal is converted to the asynchronous clear signal after the conversion. If synchronous clear behavior must be maintained, register the `sclr` signal before connecting to the `aclr0` port directly.

Example 22 shows the converted VHDL code in the Quartus II software (ALTMULT_ADD).

Example 22. Xilinx Multiplier Core to ALTMULT_ADD Megafunction Conversion—Converted VHDL Code in the Quartus II Software

```
module test(
    input clk,
    input [17:0] a,
    input [17:0] b,
    input ce,
    input sclr,
    output [35:0] p
    );

Mymult_add i1 (
    .clock0(clk),
    .dataa_0(a), // Bus [17 : 0]
    .datab_0(b), // Bus [17 : 0]
    .ena0(ce),
    .aclr0(sclr),
    .result(p)); // Bus [35 : 0]
endmodule

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY test IS
    port
    (
```

```
        clk :  IN  STD_LOGIC;
        a :  IN  STD_LOGIC_VECTOR(17 downto 0);
        b :  IN  STD_LOGIC_VECTOR(17 downto 0);
        ce :  IN  STD_LOGIC;
        sclr :  IN  STD_LOGIC;
        p :  OUT  STD_LOGIC_VECTOR(35 downto 0)
    );
END test;

ARCHITECTURE arch OF test IS

component mymult
    PORT(clock0 : IN STD_LOGIC;
         dataa_0 : IN STD_LOGIC_VECTOR(17 downto 0);
         datab_0 : IN STD_LOGIC_VECTOR(17 downto 0);
         ena0 : IN STD_LOGIC;
         aclr0 : IN STD_LOGIC;
         result : OUT STD_LOGIC_VECTOR(35 downto 0)
    );
end component;
BEGIN
i1 : mymult
PORT MAP(clock0 => clk,
         dataa_0 => a,
         datab_0 => b,
         ena0 => ce,
         aclr0 => sclr,
         result => p);
END;
```

Set Equivalent Xilinx Design Constraints

When designing for a Xilinx device, the User Constraint File (**.ucf**) contains the constraints and attributes for the design. The **.ucf** contains all of the design's constraints and attributes, including timing constraints and device constraints. This file is similar to the Quartus II Settings File (**.qsf**) and Synopsys Design Constraints File (**.sdc**), in which all the device and timing constraints are stored. Refer to "Design Constraints" on page 12 for various ways of entering design constraints.

Altera recommends specifying the necessary requirements for the design to function correctly. However, Xilinx-based placement constraints do not carry over to Altera placement constraints. You must not make placement constraints to a design until the conversion process involving the Quartus II software is complete.

The following Xilinx-based placement constraints cannot be carried forward to the Altera Quartus II software:

- LOC
- RLOC
- RLOC_ORIGIN
- RLOC_RANGE
- MAP

Device Constraints

Table 18 summarizes the most common Xilinx device constraints and Altera equivalent device constraints in assignment name and QSF variable. To set a constraint, you can directly modify the **.qsf** or use the Quartus II Assignment Editor.

Table 18. Altera Equivalent Device Constraints

Xilinx Constraints	Altera Constraint (Assignment Name)	Altera Constraints (QSF Variable)	Description
DRIVE	Current Strength	CURRENT_STRENGTH_NEW	Controls the output pin current value
FAST	Slew Rate	SLEW_RATE (1)	Turns on Fast Slew Rate Control.
IOB	Fast Input Register Fast Output Register	FAST_INPUT_REGISTER FAST_OUTPUT_REGISTER	Specifies whether or not a register should be placed in the IOB of the device
IOSTANDARD	IO Standard	IO_STANDARD	Specifies the I/O standard for an I/O pin
KEEP	Implement as Output of Logic Cell	"attribute keep" (VHDL) "synthesis keep" (Verilog)	Prevents a net from either being absorbed by a block or synthesized out

Note to Table 18:

(1) SLEW_RATE is used for the Cyclone III, Stratix III, and Stratix IV device families.

DRIVE

Equivalent to the DRIVE constraint in the Xilinx ISE software, CURRENT_STRENGTH_NEW is a logic option that sets the drive strength of a pin. This option must be assigned to an output or bidirectional pin or it is ignored

The following example shows how to set the equivalent DRIVE constraint with 12 mA to the output "q1".

Example of UCF command:

```
# Set drive strength 12 mA to q1
INST "q1" DRIVE=12;
```

Equivalent QSF command:

```
# Set drive strength 12 mA to q1
set_instance_assignment -name CURRENT_STRENGTH_NEW 12MA -to q1
```

For more information about the current strength feature in the device, refer to the specific device handbook and the Quartus II Help.

FAST

Equivalent to the FAST constraint in the Xilinx ISE software, SLEW_RATE is a logic option that implements control of low-to-high or high-to-low transitions on output pins to help reduce switching noise. When a large number of output pins switch simultaneously, pins that use the lower SLEW_RATE option help reduce switching noise. This option is only applicable to output or bidirectional pins.

The following example shows how to set the equivalent FAST constraint to the output "q1".

Example of UCF command:

```
# set fast slew rate to q1
```

```
NET "q1" FAST;
```

Equivalent QSF command:

```
# set programmable slew rate to q1
set_instance_assignment -name SLEW_RATE 2 -to q1
```

For more information about the slew rate feature in the device, refer to the specific device handbook.

IOB

Equivalent to the `IOB` constraint in Xilinx, the `FAST_INPUT_REGISTER` and `FAST_OUTPUT_REGISTER` logic options implement an input register and output register in an I/O cell that has a fast, direct connection from an I/O pin.

The following example shows how to set the equivalent `IOB` constraint to the input "`d1`" or the output "`q1`".

Example of UCF command:

```
# Set IOB to input d1
INST "d1" IOB=TRUE;

# Set IOB to output q1
INST "q1" IOB=TRUE;
```

Equivalent QSF command:

```
# Set FAST_INPUT_REGSITER to input d1
set_instance_assignment -name FAST_INPUT_REGISTER ON -to d1

# Set FAST_OUTPUT_REGSITER to output q1
set_instance_assignment -name FAST_OUTPUT_REGISTER ON -to q1
```

For more information about the slew fast input and output register features in the device, refer to the specific device handbook and the Quartus II Help.

IOSTANDARD

Equivalent to the `IOSTANDARD` constraint in Xilinx, the `IO_STANDARD` logic option uniquely defines the input and output (VCCIO) voltage, reference VREF voltage (if applicable), and the types of input and output buffers used for I/O pins.

The following example shows how to set the equivalent `IOSTANDARD` constraint (Differential SSTL-2 Class I) to the "`q2`" output.

Example of UCF command:

```
# Set Differential SSTL-2 Class 1 I/O Standard to q2
NET "q2" IOSTANDARD = SSTL2_I;
```

Equivalent QSF command:

```
# Set Differential SSTL-2 Class 1 I/O Standard to q2
set_instance_assignment -name IO_STANDARD "SSTL-2 CLASS I" -to q2
```

KEEP

Equivalent to the `KEEP` constraints, Attribute Keep (VHDL) or Synthesis Keep (Verilog) in the Quartus II software are logic options that direct the Compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. Similarly, you can also can set the **Implement as Output of Logic Cell** logic option in the Quartus II Assignment Editor.

The following example shows how both VHDL and Verilog HDL to set the equivalent `KEEP` constraint (Differential SSTL-2 Class I) to `my_wire` signal.

Verilog HDL example in the ISE software:

```
(* KEEP = "TRUE" *) wire my_wire
```

Equivalent Verilog HDL example in the Quartus II software:

```
(   * keep = 1 *) wire my_wire;
```

VHDL example in the ISE software:

```
signal my_wire: bit;
attribute keep : string;
attribute keep of my_wire: signal is "TRUE";
```

Equivalent VHDL example in the Quartus II software:

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

For more information about using the attribute/synthesis keep assignment, refer to the *Quartus II Integrated Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Timing Constraints

In the Xilinx TRACE timing analyzer, global or specific timing constraints defined in the UCF are converted to SDC commands to use with the Quartus II TimeQuest timing analyzer.

Table 19 summarizes the most common Xilinx TRACE timing constraints and the equivalent Altera TimeQuest timing analyzer SDC timing constraints. You can modify the **.sdc** file or use the TimeQuest timing analyzer GUI to set the constraints.

Table 19. Xilinx TRACE versus Quartus II TimeQuest SDC Timing Constraints (Part 1 of 2)

Xilinx TRACE Timing Constraints	TimeQuest SDC Command	Description

`TIMESPEC PERIOD`	`create_clock`
	`create_generated_clock`
	`derived_pll_clocks`
		Defines all the clocks and their relationship in a design.

`OFFSET IN BEFORE`	`set_input_delay`	Input timing constraint used to define the Pad to Setup timing requirement in a design.
`OFFSET OUT AFTER`	`set_output_delay`	Output timing constraint used to define the global Clock to Pad timing requirement in a design.

Table 19. Xilinx TRACE versus Quartus II TimeQuest SDC Timing Constraints (Part 2 of 2)

Xilinx TRACE Timing Constraints	TimeQuest SDC Command	Description
FROM PADS TO PADS	set_max_delay (1)	Combinational path that constrains all combinational pin to pin paths.
TIG	set_false_path	Eliminates the paths from timing consideration during Place and route and timing analysis.

Note to Table 19:

(1) The `set_max_delay` value must take into account input and output delays. Refer to “FROM PADS TO PADS” on page 71 for more information.

TIMESPEC PERIOD

`TIMESPEC PERIOD` defines each clock in the design and as well as the relationship between the clocks. Equivalent to the `TIMESPEC PERIOD` command, the SDC commands `create_clock` and `create_generated_clock` are used by the TimeQuest timing analyzer to define the clocks and the relationship between the clocks.

The following example shows how to set the equivalent `TIMESPEC PERIOD` constraint in the SDC file.

Example of UCF command:

```
# Define clk1
NET "clk1" TNM_NET = clk1;
TIMESPEC TS_clk1 = PERIOD "clk1" 100 ns;

# Define clk2 and its relationship to clk1
NET "clk2" TNM_NET = clk2;
TIMESPEC TS_clk2 = PERIOD "clk2" "TS_clk1" * 2 PHASE + 2.5ns;
```

Equivalent SDC command:

```
# Define clk1
create_clock -name {TS_clk1} -period 100 [get_ports {clk1}]

# Define clk2 and its relationship to clk1
create_generated_clock -name {TS_clk2} -source [get_ports {clk1}]
 -divide_by 2 -offset 2.500 -master_clock {TS_clk1} [get_ports
{clk2}]
```

`clk2` is two times slower than `clk1` and has 2.5 ns phase shift. In SDC, the `-multiply_by` and `-divide_by` switches are referenced to the frequency rather than the period. Therefore, when converted to the SDC command, the `-divide_by` switch is used instead of the `-multiply_by` switch.

If you have clock definitions for DLL, DCM, or PLL clocks, you can replace them with the SDC command `derive_pll_clocks`.

The following shows how to set the equivalent DCM clock’s output definition with the `derive_pll_clocks` SDC command. This command is specific to the TimeQuest timing analyzer only.

Example of UCF (DCM) command:

```
# Generated clock definitions for DCM:
NET "clock0" TNM_NET = "clock0";
TIMESPEC "TS_clock0" = PERIOD "TS_clock_in" * 1;
NET "clock2x180" TNM_NET = "clock2x180";
TIMESPEC "TS_clock2x180" = PERIOD "TS_clock_in" / 2 PHASE + 7.50 ns;
```

Equivalent SDC (PLL) command:

```
# Create the generated clocks for the PLL.
derive_pll_clocks
```

OFFSET IN BEFORE

`OFFSET IN BEFORE` defines the Pad to Setup timing requirement. Equivalent to `OFFSET IN BEFORE`, the SDC command `set_input_delay` is used in the TimeQuest timing analyzer to define Pad to Setup timing constraints.

The following example shows how to set the equivalent `OFFSET IN BEFORE` constraint in the SDC file.

Example of UCF command:

```
#Set Pad to Setup Timing Requirement
OFFSET = IN 2.5 ns BEFORE "clock_in";
```

Equivalent SDC command:

```
#Set Pad to Setup Timing Requirement
set_input_delay -from [get_clocks clock_in] -to [all_inputs] 2.5
```

This example sets a 2.5 ns pad to set up timing constraints for all the external input pin of the device to the registers that are clocked by the `clock_in` signal.

OFFSET OUTPUT AFTER

`OFFSET OUTPUT AFTER` defines the Clock to Pad timing requirement. Equivalent to `OFFSET OUTPUT AFTER`, the SDC command `set_output_delay` is used in the TimeQuest timing analyzer to define Clock to Pad timing constraint.

The following example shows how to set the equivalent `OFFSET OUTPUT AFTER` constraint in the SDC file.

Example of UCF command:

```
#Set Clock to Pad Timing Requirement
OFFSET = OUT 10 ns AFTER "clock_in";
```

Equivalent SDC command:

```
#Set Clock to Pad Timing Requirement
set_output_delay -from [get_clocks clock_in] -to [all_outputs] 10
```

This example sets 10-ns Clock to Pad timing constraints for the `clock_in` signal to all output pins of the device.

FROM PADS TO PADS

`FROM PADS TO PADS` globally constrains all pin-to-pin paths. Equivalent to the `FROM PADS TO PADS` constraint, the SDC command `set_max_delay` is used in the TimeQuest timing analyzer to define pin-to-pin timing constraints.

The following example shows how to set the equivalent `FROM PADS TO PADS` constraint in the SDC file.

Example of UCF command:

```
#Set Pad to Pad Timing Constraint
TIMESPEC "TS_P2P" = FROM "PADS" TO "PADS" 10 ns;
```

Equivalent SDC command:

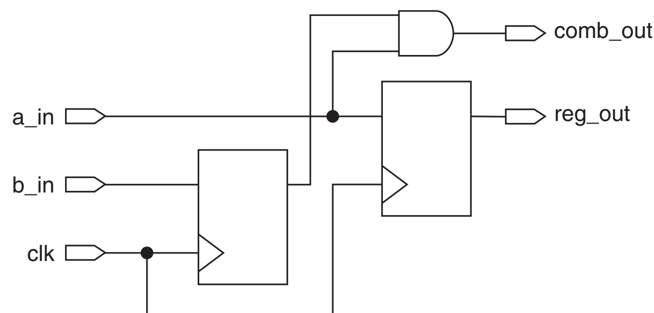
```
# Set Pad to Pad Timing Constraint
set_max_delay -from [all_inputs] -to [all_outputs] 10
```

This example sets 10 ns timing constraints from all input pins to output pins of the device.

You can use the `set_max_delay` command in the TimeQuest timing analyzer as an equivalent constraint if you account for input and output delays. The `FROM PADS TO PADS` constraint does not account for input and output delays, but the `set_max_delay` exception does. Therefore, you must modify the `set_max_delay` value to account for input and output delays.

For example, in Figure 19, the path from `a_in` to `comb_out` is affected by the input maximum delay relative to `clk` and output maximum delay on `comb_out` relative to `clk`.

Figure 19. Pin-to-Pin Timing Constraint



The slack is equal to:

<set_max_delay value from a_in to comb_out> – *<input delay>* – *<output delay>* – *<path delay from a_in to comb_out>*

For more information about input delay and output delay, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in *volume 3* of the *Quartus II Handbook*.

TIG

The `TIG` constraint eliminates paths from timing consideration during place-and-route and timing analysis. Equivalent to the `TIG` constraint, the SDC command `set_false_path` is used in the TimeQuest timing analyzer to specify a false-path exception, removing (or cutting) paths from timing analysis.

You can set false paths by net, instance, or pin. The following example shows how to set the equivalent `TIG` constraint in the SDC file.

Example of UCF command:

```
# Define false paths for all paths that pass through a particular net
NET "net_name" TIG;

# Define false paths for all paths that pass through a particular pin
PIN "ff_inst.RST" TIG=TS_1;

# Define false paths for all paths that pass through a particular
# instance
INST "instance_name" TIG=TS_2;
```

Equivalent SDC command:

```
# Define false paths for all paths that pass through a particular net
set_false_path -through [get_nets {net_name}]

# Define false paths for all paths that pass through a particular pin
set_false_path -through [get_pins {ff_inst|RST}]

# Define false paths for all paths that pass through a particular
instance
set_false_path -through [get_pins {instance_name|*}]
```

The `-through` switch does not allow you to specify the instance name directly; therefore, the wildcard is being used to specify all the input and outputs paths of the instance name.

For more information about using the SDC commands and their usage, refer to the *The Quartus II TimeQuest Timing Analyzer* chapter *volume 3* of the *Quartus II Handbook*.

Conclusion

The Quartus II design software provides a complete design environment that you can easily adapt to your design for the development of Altera FPGAs, CPLDs, and HardCopy ASIC devices.

Programmable logic design and compilation flow is very similar between Altera Quartus II software and Xilinx ISE software, and in most cases, your ISE design is easily imported into the Quartus II software design environment. You can improve your design conversion experience by following the design conversion guidelines and considerations discussed in this application note, including migrating a design targeted at a Xilinx device to one that is compatible with an Altera device.

Document Revision History

Table 20 lists the revision history for this document.

Table 20. Document Revision History

Date	Version	Changes
March 2013	7.0	■ Revised content for software versions ISE 14.2 and Quartus II 12.1 ■ Removed outdated design examples. ■ Updated template
November 2009	6.2	Corrected `set_max_delay` constraint equivqalents for `OFFSET IN BEFORE` and `OFFSET OUTPUT AFTER` UCF commands in Timing Constraints section.
April 2009	v6.2	Added Appendix A: Design Example and Appendix B
July 2008	v6.0	Revised and restructured content for software versions ISE 10.1 and Quartus II 8.0
June 2005	v5.0	■ Revised content for software versions ISE 7.1 and Quartus II 5.0 ■ Updated terminology ■ Added Pin Planner subsection ■ Added Quartus II Incremental Compilation
February 2004	4.0	■ Revised content for software versions ISE 6.3i and Quartus II 4.2 ■ Updated Table 6 for Power ■ Updated cross-probing chart
January 2004	3.1	Updated terminology
October 2003	3.0	■ Revised content for software versions ISE 6.2i and Quartus II 4.1 sp2 ■ Added information on cross-probing
July 2003	2,0	■ Revised content for software versions ISE 5.1i and Quartus II 3.0 ■ Added information on the Quartus II modular executables and command-line scripting ■ Added information on DDR RAM conversions
November 2002	1.0	Initial release.