Quartus II Handbook Volume 1: Design and Synthesis

Subscribe

Send Feedback

QII5V1
2015.05.04

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

Managing Quartus II Projects

1

2015.05.04

QII5V1

Subscribe

Send Feedback

The Quartus II software organizes and manages the elements of your design within a *project*. The project encapsulates information about your design hierarchy, libraries, constraints, and project settings. Click **File** > **New Project Wizard** to quickly create a new project and specify basic project settings

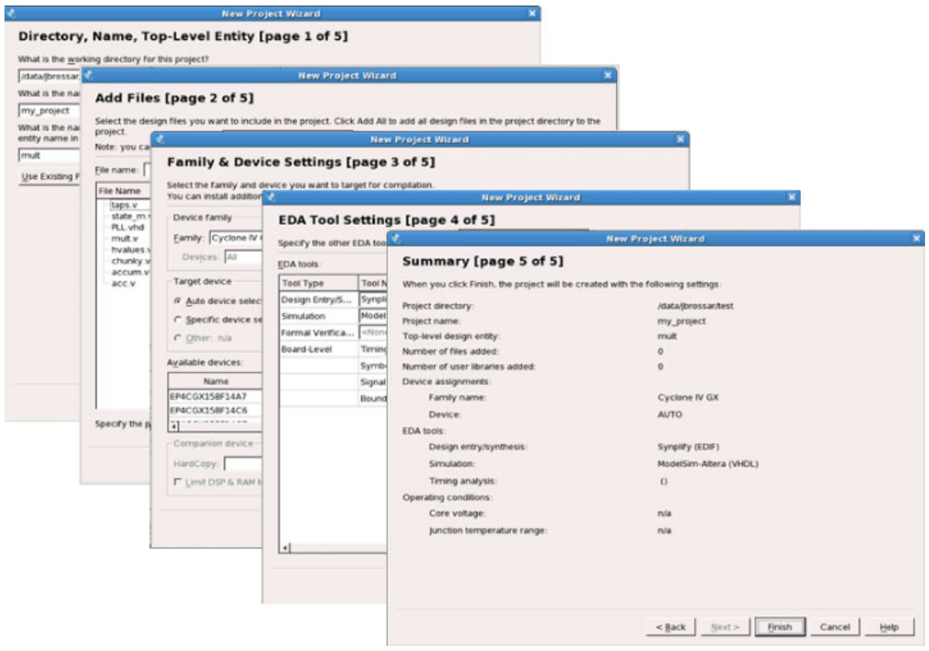
When you open a project, a unified GUI displays integrated project information. The Project Navigator allows you to view and edit the elements of your project. The Messages window lists important information about project processing.

You can save multiple revisions of your project to experiment with settings that achieve your design goals. Quartus II projects support team-based, distributed work flows and a scripting interface.

Quick Start

To quickly create a project and specify basic settings, click **File** > **New Project Wizard**.

New Project Wizard



New Project Wizard
Directory, Name, Top-Level Entity [page 1 of 5]
Add Files [page 2 of 5]
Family & Device Settings [page 3 of 5]
EDA Tool Settings [page 4 of 5]
< Back
Next >
Finish
Cancel
Help

Summary [page 5 of 5]

When you click Finish, the project will be created with the following settings:

Project directory:	/data/jbrossar/test
Project name:	my_project
Top-level design entity:	mult
Number of files added:	0
Number of user libraries added:	0
Device assignments:	
Family name:	Cyclone IV GX
Device:	AUTO
EDA tools:	
Design entry/synthesis:	Synplify (EDIF)
Simulation:	ModelSim-Altera (VHDL)
Timing analysis:	()
Operating conditions:	
Core voltage:	n/a
Junction temperature range:	n/a

Note: The New Project Wizard offers project templates based on fully functioning design examples. Select the **Project template** option to choose a project template that is ready to compile. Altera provides additional project templates as available.

Understanding Quartus II Projects

A single Quartus II Project File (**.qpf**) represents each project. The text-based **. qpf** references the Quartus II Settings File (**.qsf**), that lists all project files and stores project and entity settings. When you make project changes in the GUI, these text files automatically store the changes. The GUI helps to manage:

- Design, EDA, IP core, and Qsys system files
- Project settings and constraint files
- Project archive and migration files

Table 1-1: Quartus II Project Files

File Type	Contains	To Edit	Format
Project file	Project and revision name	**File** > **New Project Wizard**	Quartus II Project File (**.qpf**)
Project settings	Lists design files, entity settings, target device, synthesis directives, placement constraints	**Assignments** > **Settings**	Quartus II Settings File (**.qsf**)
Project database	Compilation results	**Project** > **Export Database**	Exported Partition File (**.qxp**)
Timing constraints	Clock properties, exceptions, setup/hold	**Tools** > **TimeQuest Timing Analyzer**	Synopsys Design Constraints File (**.sdc**)
Logic design files	RTL and other design source files	**File** > **New**	All supported HDL files
Program-ming files	Device programming image and information	**Tools** > **Programmer**	SRAM Object File (**.sof**) Programmer Object File (**.pof**)
Project library	Project and global library information	**Tools** > **Options** > **Libraries**	**.qsf**(project) **quartus2.ini** (global)
IP core files	IP core logic, synthesis, and simulation information	**Tools** > **IP Catalog**	All supported HDL files Quartus II IP File (**.qip**)
Qsys system files	Qsys system and IP core files	**Tools** > **Qsys**	Qsys System File (**.qsys**)

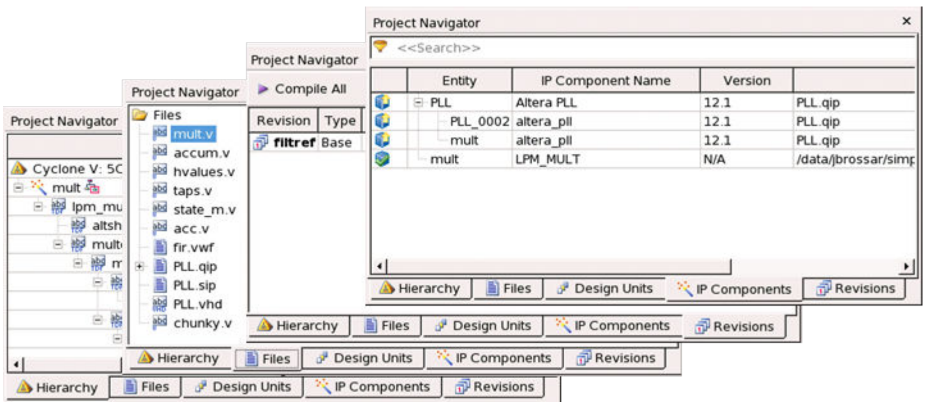
File Type	Contains	To Edit	Format
EDA tool files	Generated for third-party EDA tools	**Tools** > **Options** > **EDA Tool Options**	Verilog Output File (**.vo**) VHDL Output File (**.vho**) Verilog Quartus Mapping File (**.vqm**)
Archive files	Complete project as single compressed file	**Project** > **Archive Project**	Quartus II Archive File (**.qar**)

Viewing Basic Project Information

View basic information about your project in the Project Navigator, Report panel, and Messages window. View project elements in the Project Navigator (**View** > **Utility Windows** > **Project Navigator**). The Project Navigator displays key project information, including design files, IP components, and revisions of your project. Use the Project Navigator to:

- View and modify the design hierarchy (**right-click** > **Set as Top-Level Entity**)
- Set the project revision (**right-click** > **Set Current Revision**)
- View and update logic design files and constraint files (**right-click** > **Open**)
- Update IP component version information (**right-click** > **Upgrade IP Component**)

Figure 1-1: Project Navigator Hierarchy, Files, Revisions, and IP



	Entity	IP Component Name	Version	
	PLL	Altera PLL	12.1	PLL.qip
	PLL_0002	altera_pll	12.1	PLL.qip
	mult	altera_pll	12.1	PLL.qip
	mult	LPM_MULT	N/A	/data/jbrossar/simp

Revision	Type
filtref	Base

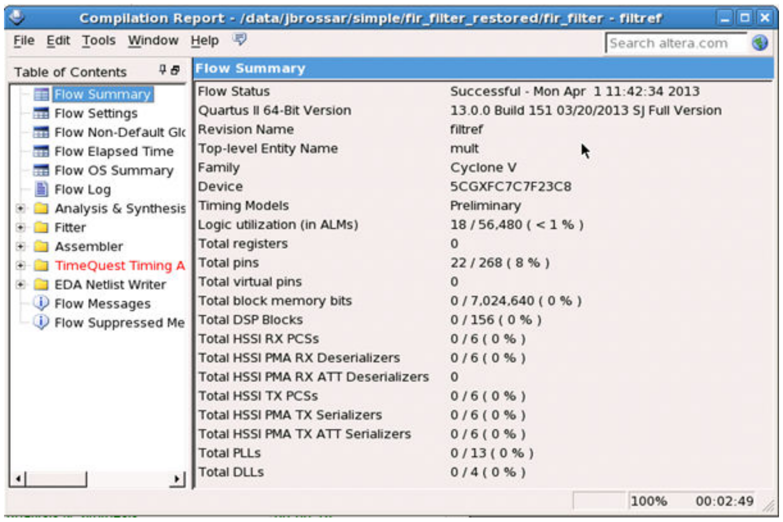
Viewing Project Reports

The Report panel (**Processing** > **Compilation Report**) displays detailed reports after project processing, including the following:

- Analysis & Synthesis reports
- Fitter reports
- Timing analysis reports
- Power analysis reports
- Signal integrity reports

Analyze the detailed project information in these reports to determine correct implementation. Right-click report data to locate and edit the source in project files.

Figure 1-2: Report Panel



Flow Summary

Flow Status	Successful - Mon Apr 1 11:42:34 2013
Quartus II 64-Bit Version	13.0.0 Build 151 03/20/2013 SJ Full Version
Revision Name	filtref
Top-level Entity Name	mult
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Preliminary
Logic utilization (in ALMs)	18 / 56,480 (< 1 %)
Total registers	0
Total pins	22 / 268 (8 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI PMA RX ATT Deserializers	0
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total HSSI PMA TX ATT Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Related Information

List of Compilation Reports

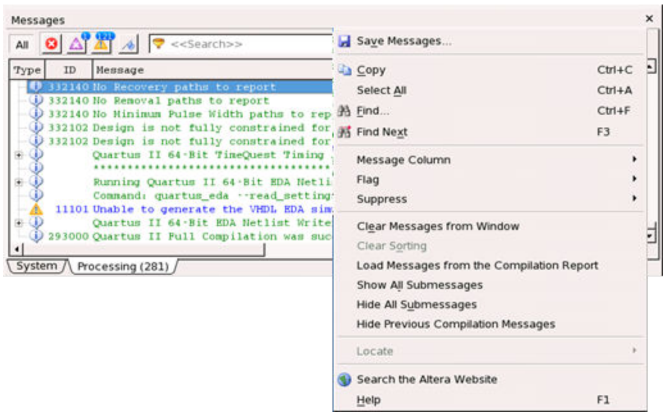
Viewing Project Messages

The Messages window (**View** > **Utility Windows** > **Messages**) displays information, warning, and error messages about Quartus II processes. Right-click messages to locate the source or get message help.

- **Processing** tab—displays messages from the most recent process
- **System** tab—displays messages unrelated to design processing
- **Search**—locates specific messages

Messages are written to `stdout` when you use command-line executables.

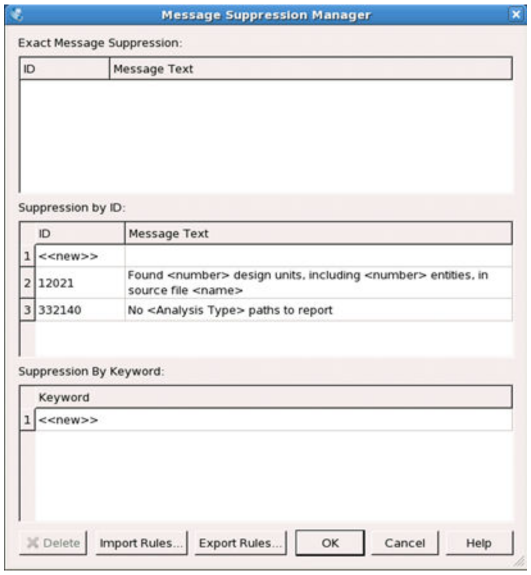
Figure 1-3: Messages Window



```
Type  ID     Message
      332140 No Recovery paths to report
      332140 No Removal paths to report
      332140 No Minimum Pulse Width paths to rep
      332102 Design is not fully constrained for
      332102 Design is not fully constrained for
             Quartus II 64-Bit TimeQuest Timing
             ********************************
             Running Quartus II 64-Bit EDA Netli
             Command: quartus_eda --read_setting
      11101  Unable to generate the VHDL EDA sim
             Quartus II 64-Bit EDA Netlist Write
      293000 Quartus II Full Compilation was suc
```

You can suppress display of unimportant messages so they do not obscure valid messages.

Figure 1-4: Message Suppression by Message ID Number



Message Suppression Manager

Exact Message Suppression:

ID	Message Text

Suppression by ID:

	ID	Message Text
1	<<new>>	
2	12021	Found <number> design units, including <number> entities, in source file <name>
3	332140	No <Analysis Type> paths to report

Suppression By Keyword:

	Keyword
1	<<new>>

Suppressing Messages

To supress messages, right-click a message and choose any of the following:

- **Suppress Message**—suppresses all messages matching exact text
- **Suppress Messages with Matching ID**—suppresses all messages matching the message ID number, ignoring variables
- **Suppress Messages with Matching Keyword**—suppresses all messages matching keyword or hierarchy

Message Suppression Guidelines

- You cannot suppress error or Altera legal agreement messages.
- Suppressing a message also suppresses any submessages.
- Message suppression is revision-specific. Derivative revisions inherit any suppression.
- You cannot edit messages or suppression rules during compilation.

Managing Project Settings

The New Project Wizard helps you initially assign basic project settings. Optimizing project settings enables the Compiler to generate programming files that meet or exceed your specifications.

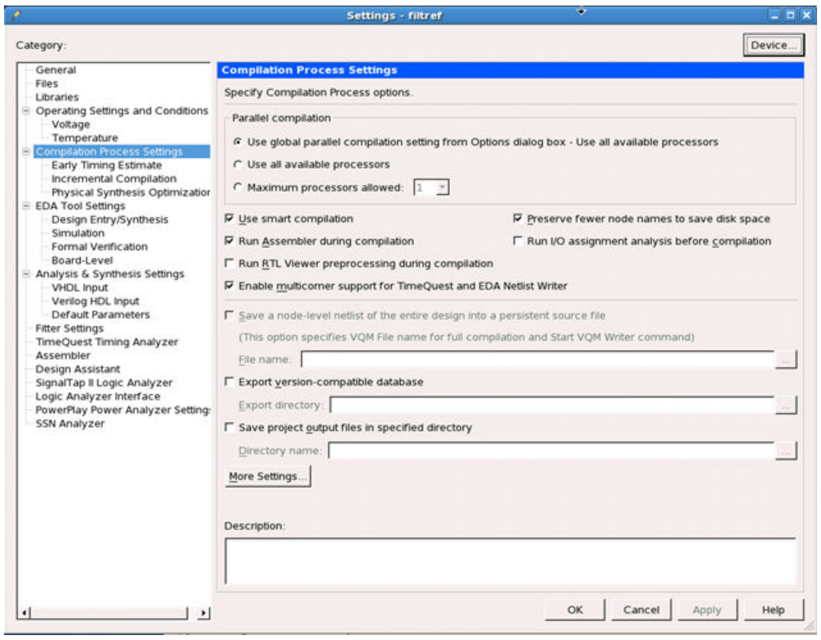
The **.qsf** stores each revision's project settings.

Click **Assignments** > **Settings** to access global project settings, including:

- Project files list
- Synthesis directives and constraints
- Logic options and compiler effort levels
- Placement constraints
- Timing constraint files
- Operating temperature limits and conditions
- File generation for other EDA tools
- Target device (click **Assignments** > **Device**)

The Quartus II Default Settings File (<*revision name*>**_assignment_defaults.qdf**) stores initial settings and constraints for each new project revision.

Figure 1-5: Settings Dialog Box for Global Project Settings



Compilation Process Settings

Specify Compilation Process options.

Parallel compilation

- [✓ Use global parallel compilation setting from Options dialog box - Use all available processors]
- Use all available processors
- Maximum processors allowed: 1

- [✓] Use smart compilation
- [✓] Run Assembler during compilation
- [ ] Run RTL Viewer preprocessing during compilation
- [✓] Enable multicorner support for TimeQuest and EDA Netlist Writer
- [✓] Preserve fewer node names to save disk space
- [ ] Run I/O assignment analysis before compilation
- [ ] Save a node-level netlist of the entire design into a persistent source file

(This option specifies VQM File name for full compilation and Start VQM Writer command)

File name:

- [ ] Export version-compatible database

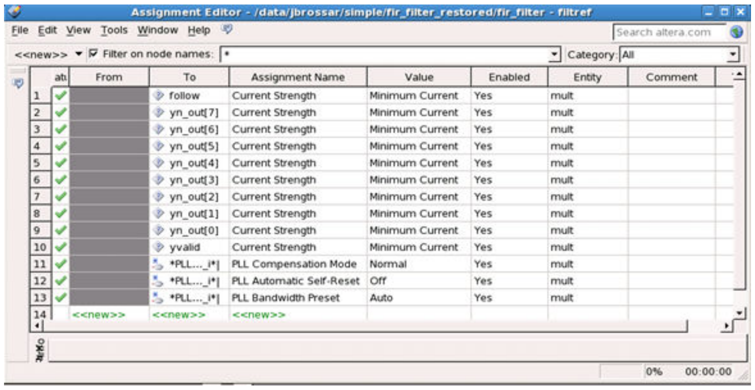
Export directory:

- [ ] Save project output files in specified directory

Directory name:

The Assignment Editor (**Tools** > **Assignment Editor**) provides a spreadsheet-like interface for assigning all instance-specific settings and constraints.

Figure 1-6: Assignment Editor Spreadsheet



Assignment Editor - /data/jbrossar/simple/fir_filter_restored/fir_filter - filtref

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

	at	From	To	Assignment Name	Value	Enabled	Entity	Comment
1	✓		follow	Current Strength	Minimum Current	Yes	mult	
2	✓		yn_out[7]	Current Strength	Minimum Current	Yes	mult	
3	✓		yn_out[6]	Current Strength	Minimum Current	Yes	mult	
4	✓		yn_out[5]	Current Strength	Minimum Current	Yes	mult	
5	✓		yn_out[4]	Current Strength	Minimum Current	Yes	mult	
6	✓		yn_out[3]	Current Strength	Minimum Current	Yes	mult	
7	✓		yn_out[2]	Current Strength	Minimum Current	Yes	mult	
8	✓		yn_out[1]	Current Strength	Minimum Current	Yes	mult	
9	✓		yn_out[0]	Current Strength	Minimum Current	Yes	mult	
10	✓		yvalid	Current Strength	Minimum Current	Yes	mult	
11	✓		*PLL..._i*|	PLL Compensation Mode	Normal	Yes	mult	
12	✓		*PLL..._i*|	PLL Automatic Self-Reset	Off	Yes	mult	
13	✓		*PLL..._i*|	PLL Bandwidth Preset	Auto	Yes	mult	
14		<<new>>	<<new>>	<<new>>				

0% 00:00:00

Optimizing Project Settings

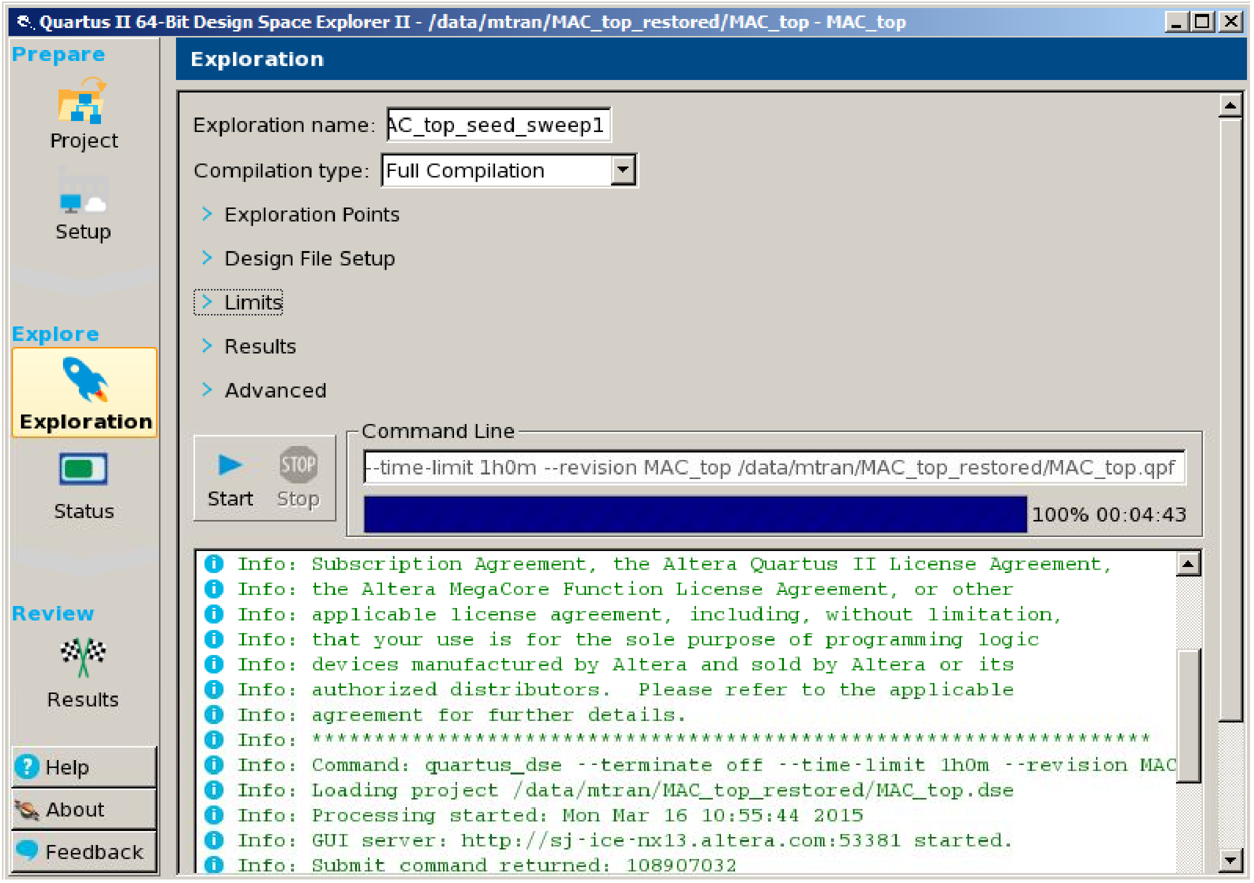
Optimize project settings to meet your design goals. The Quartus II Design Space Explorer II iteratively compiles your project with various setting combinations to find the optimal setting for your goals. Alternatively, you can create a project revision or project copy to manually compare various project settings and design combinations.

Optimizing with Design Space Explorer II

Use Design Space Explorer II (**Tools** > **Launch Design Space Explorer II**) to find optimal project settings for resource, performance, or power optimization goals. Design Space Explorer II (DSE II) processes your design using various setting and constraint combinations, and reports the best settings for your design.

DSE II attempts multiple seeds to identify one meeting your requirements. DSE II can run different compilations on multiple computers in parallel to streamline timing closure.

Figure 1-7: Design Space Explorer II



Exploration name: AC_top_seed_sweep1
Compilation type: Full Compilation

Command Line: --time-limit 1h0m --revision MAC_top /data/mtran/MAC_top_restored/MAC_top.qpf

100% 00:04:43

```
Info: Subscription Agreement, the Altera Quartus II License Agreement,
Info: the Altera MegaCore Function License Agreement, or other
Info: applicable license agreement, including, without limitation,
Info: that your use is for the sole purpose of programming logic
Info: devices manufactured by Altera and sold by Altera or its
Info: authorized distributors.  Please refer to the applicable
Info: agreement for further details.
Info: *******************************************************************
Info: Command: quartus_dse --terminate off --time-limit 1h0m --revision MAC
Info: Loading project /data/mtran/MAC_top_restored/MAC_top.dse
Info: Processing started: Mon Mar 16 10:55:44 2015
Info: GUI server: http://sj-ice-nx13.altera.com:53381 started.
Info: Submit command returned: 108907032
```

Optimizing with Project Revisions

You can save multiple, named project revisions within your Quartus II project (**Project** > **Revisions**).

Each revision captures a unique set of project settings and constraints, but does not capture any logic design file changes. Use revisions to experiment with different settings while preserving the original.You can compare revisions to determine the best combination, or optimize different revisions for various applications. Use revisions for the following:

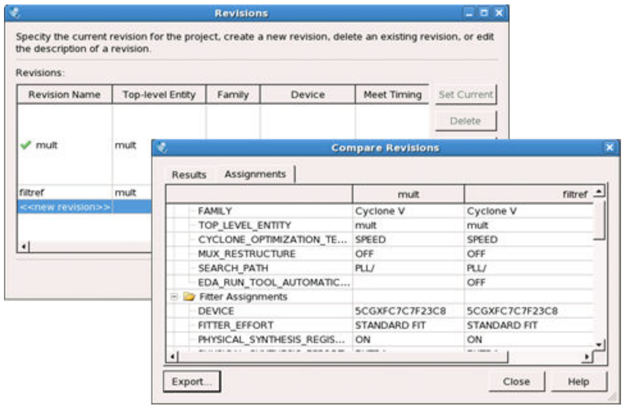
- Create a unique revision to optimize a design for different criteria, such as by area in one revision and by f_{MAX} in another revision.
- When you create a new revision the default Quartus II settings initially apply.
- Create a revision of a revision to experiment with settings and constraints. The child revision includes all the assignments and settings of the parent revision.

You create, delete, specify current, and compare revisions in the **Revisions** dialog box. Each time you create a new project revision, the Quartus II software creates a new **.qsf** using the revision name.

To compare each revision's synthesis, fitting, and timing analysis results side-by-side, click **Project** > **Revisions** and then click **Compare**.

In addition to viewing the compilation results of each revision, you can also compare the assignments for each revision. This comparison reveals how different optimization options affect your design.

Figure 1-8: Comparing Project Revisions



Specify the current revision for the project, create a new revision, delete an existing revision, or edit the description of a revision.

Revisions:

Revision Name	Top-level Entity	Family	Device	Meet Timing
✓ mult	mult			
filtref	mult			
<<new revision>>				

	mult	filtref
FAMILY	Cyclone V	Cyclone V
TOP_LEVEL_ENTITY	mult	mult
CYCLONE_OPTIMIZATION_TE...	SPEED	SPEED
MUX_RESTRUCTURE	OFF	OFF
SEARCH_PATH	PLL/	PLL/
EDA_RUN_TOOL_AUTOMATIC...		OFF
Fitter Assignments		
DEVICE	5CGXFC7C7F23C8	5CGXFC7C7F23C8
FITTER_EFFORT	STANDARD FIT	STANDARD FIT
PHYSICAL_SYNTHESIS_REGIS...	ON	ON

Copying Your Project

Click **Project** > **Copy Project** to create a separate copy of your project, rather than just a revision within the same project.

The project copy includes all design files, **.qsf**(s), and project revisions. Use this technique to optimize project copies for different applications. For example, optimize one project to interface with a 32-bit data bus, and optimize a project copy to interface with a 64-bit data bus.

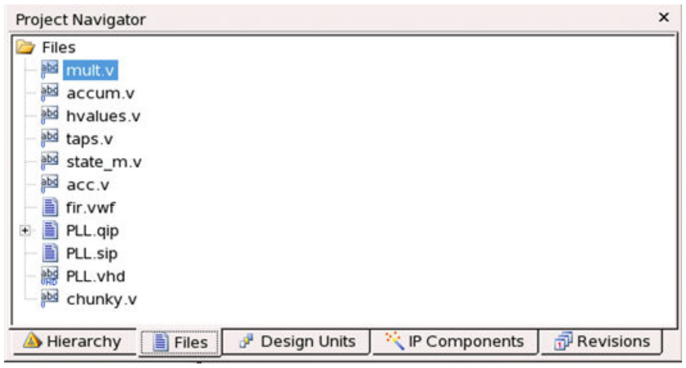
Managing Logic Design Files

The Quartus II software helps you create and manage the logic design files in your project. Logic design files contain the logic that implements your design. When you add a logic design file to the project, the Compiler automatically compiles that file as part of the project. The Compiler synthesizes your logic design files to generate programming files for your target device.

The Quartus II software includes full-featured schematic and text editors, as well as HDL templates to accelerate your design work. The Quartus II software supports VHDL Design Files (**.vhd**), Verilog HDL Design Files (**.v**), SystemVerilog (**. sv**) and schematic Block Design Files (**. bdf**). The Quartus II software also supports Verilog Quartus Mapping (**.vqm**) design files generated by other design entry and synthesis tools. In addition, you can combine your logic design files with Altera and third-party IP core design files, including combining components into a Qsys system (**. qsys**).

The New Project Wizard prompts you to identify logic design files. Add or remove project files by clicking **Project** > **Add/Remove Files in Project.** View the project's logic design files in the Project Navigator.

Figure 1-9: Design and IP Files in Project Navigator



Right-click files in the Project Navigator to:

- **Open** and edit the file
- **Remove File from Project**
- **Set as Top-Level Entity** for the project revision
- **Create a Symbol File for Current File** for display in schematic editors
- Edit file **Properties**

Including Design Libraries

You can include design files libraries in your project. Specify libraries for a single project, or for all Quartus II projects. The**.qsf** stores project library information.

The **quartus2.ini** file stores global library information.

Related Information

Design Library Migration Guidelines on page 1-41

Specifying Design Libraries

To specify project libraries from the GUI:

1. Click **Assignment** > **Settings**.
2. Click **Libraries** andspecify the **Project Library name** or **Global Library name**.Alternatively, you can specify project libraries with `SEARCH_PATH` in the **.qsf**, and global libraries in the **quartus2.ini** file.

Related Information

- Recommended Design Practices on page 11-1
- Recommended HDL Coding Styles on page 12-1

Managing Timing Constraints

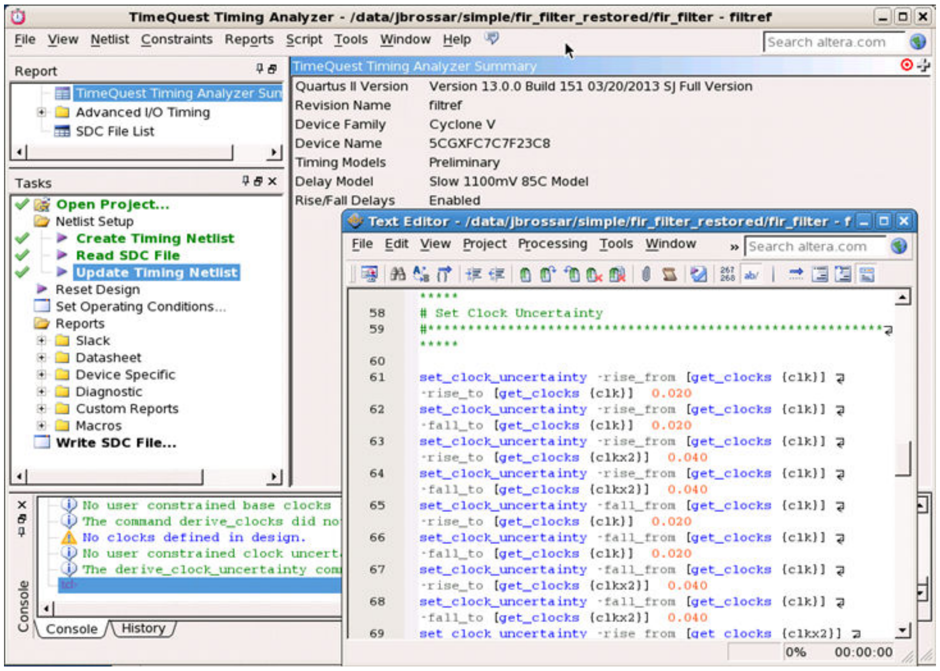
View basic information about your project in the Project Navigator, Report panel, and Messages window.

Apply appropriate timing constraints to correctly optimize fitting and analyze timing for your design. The Fitter optimizes the placement of logic in the device to meet your specified timing and routing constraints.

Specify timing constraints in the TimeQuest Timing Analyzer (**Tools** > **TimeQuest Timing Analyzer**), or in an **.sdc** file. Specify constraints for clock characteristics, timing exceptions, and external signal setup and hold times before running analysis. TimeQuest reports the detailed information about the performance of your design compared with constraints in the Compilation Report panel.

Save the constraints you specify in the GUI in an industry-standard Synopsys Design Constraints File (**.sdc**). You can subsequently edit the text-based **.sdc** file directly.

Figure 1-10: TimeQuest Timing Analyzer and SDC Syntax Example


TimeQuest Timing Analyzer - /data/jbrossar/simple/fir_filter_restored/fir_filter - filtref
File View Netlist Constraints Reports Script Tools Window Help
Search altera.com
Report
TimeQuest Timing Analyzer Sum
Advanced I/O Timing
SDC File List
Tasks
Open Project...
Netlist Setup
Create Timing Netlist
Read SDC File
Update Timing Netlist
Reset Design
Set Operating Conditions...
Reports
Slack
Datasheet
Device Specific
Diagnostic
Custom Reports
Macros
Write SDC File...
Console
History
TimeQuest Timing Analyzer Summary
Text Editor - /data/jbrossar/simple/fir_filter_restored/fir_filter - f
File Edit View Project Processing Tools Window
0%
00:00:00

Quartus II Version	Version 13.0.0 Build 151 03/20/2013 SJ Full Version
Revision Name	filtref
Device Family	Cyclone V
Device Name	5CGXFC7C7F23C8
Timing Models	Preliminary
Delay Model	Slow 1100mV 85C Model
Rise/Fall Delays	Enabled

```
No user constrained base clocks
The command derive_clocks did no
No clocks defined in design.
No user constrained clock uncert
The derive_clock_uncertainty com
tcl>
```

```
*****
# Set Clock Uncertainty
#*************************************************************
*****

set_clock_uncertainty -rise_from [get_clocks {clk}] 
-rise_to [get_clocks {clk}]  0.020
set_clock_uncertainty -rise_from [get_clocks {clk}] 
-fall_to [get_clocks {clk}]  0.020
set_clock_uncertainty -rise_from [get_clocks {clk}] 
-rise_to [get_clocks {clkx2}]  0.040
set_clock_uncertainty -rise_from [get_clocks {clk}] 
-fall_to [get_clocks {clkx2}]  0.040
set_clock_uncertainty -fall_from [get_clocks {clk}] 
-rise_to [get_clocks {clk}]  0.020
set_clock_uncertainty -fall_from [get_clocks {clk}] 
-fall_to [get_clocks {clk}]  0.020
set_clock_uncertainty -fall_from [get_clocks {clk}] 
-rise_to [get_clocks {clkx2}]  0.040
set_clock_uncertainty -fall_from [get_clocks {clk}] 
-fall_to [get_clocks {clkx2}]  0.040
set clock uncertainty -rise from [get clocks {clkx2}] 
```

Related Information

Quartus II TimeQuest Timing Analyzer

Introduction to Altera IP Cores

Altera® and strategic IP partners offer a broad portfolio of off-the-shelf, configurable IP cores optimized for Altera devices. The Quartus® II software installation includes the Altera IP library. You can integrate optimized and verified Altera IP cores into your design to shorten design cycles and maximize performance. You can evaluate any Altera IP core in simulation and compilation in the Quartus II software. The Quartus II software also supports integration of IP cores from other sources. Use the IP Catalog to efficiently parameterize and generate synthesis and simulation files for a custom IP variation.

The Altera IP library includes the following categories of IP cores:

- Basic functions
- DSP functions
- Interface protocols
- Low power functions
- Memory interfaces and controllers
- Processors and peripherals

Note: The IP Catalog (**Tools** > **IP Catalog)** and parameter editor replace the MegaWizard™ Plug-In Manager for IP selection and parameterization, beginning in Quartus II software version 14.0. Use the IP Catalog and parameter editor to locate and paramaterize Altera and other supported IP cores.

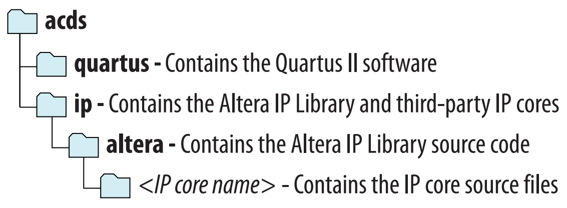
Related Information

- **IP User Guide Documentation**
- **Altera IP Release Notes**

Installing and Licensing IP Cores

The Altera IP Library provides many useful IP core functions for your production use without purchasing an additional license. Some Altera MegaCore® IP functions require that you purchase a separate license for production use. However, the OpenCore® feature allows evaluation of any Altera IP core in simulation and compilation in the Quartus II software. After you are satisfied with functionality and perfformance, visit the Self Service Licensing Center to obtain a license number for any Altera product.

Figure 1-11: IP Core Installation Path


acds
quartus - Contains the Quartus II software
ip - Contains the Altera IP Library and third-party IP cores
altera - Contains the Altera IP Library source code
<IP core name> - Contains the IP core source files

Note: The default IP installation directory on Windows is ***<drive>*:\altera\**<*version number*>; on Linux it is <*home directory*>**/altera/** <*version number*>.

Related Information

- **Altera Licensing Site**
- **Altera Software Installation and Licensing Manual**

OpenCore Plus IP Evaluation

Altera's free OpenCore Plus feature allows you to evaluate licensed MegaCore IP cores in simulation and hardware before purchase. You need only purchase a license for MegaCore IP cores if you decide to take your design to production. OpenCore Plus supports the following evaluations:

- Simulate the behavior of a licensed IP core in your system.
- Verify the functionality, size, and speed of the IP core quickly and easily.
- Generate time-limited device programming files for designs that include IP cores.
- Program a device with your IP core and verify your design in hardware.

OpenCore Plus evaluation supports the following two operation modes:

- Untethered—run the design containing the licensed IP for a limited time.
- Tethered—run the design containing the licensed IP for a longer time or indefinitely. This requires a connection between your board and the host computer.

Note: All IP cores that use OpenCore Plus time out simultaneously when any IP core in the design times out.

IP Catalog and Parameter Editor

The Quartus II IP Catalog (**Tools** > **IP Catalog**) and parameter editor help you easily customize and integrate IP cores into your project. You can use the IP Catalog and parameter editor to select, customize, and generate files representing your custom IP variation.

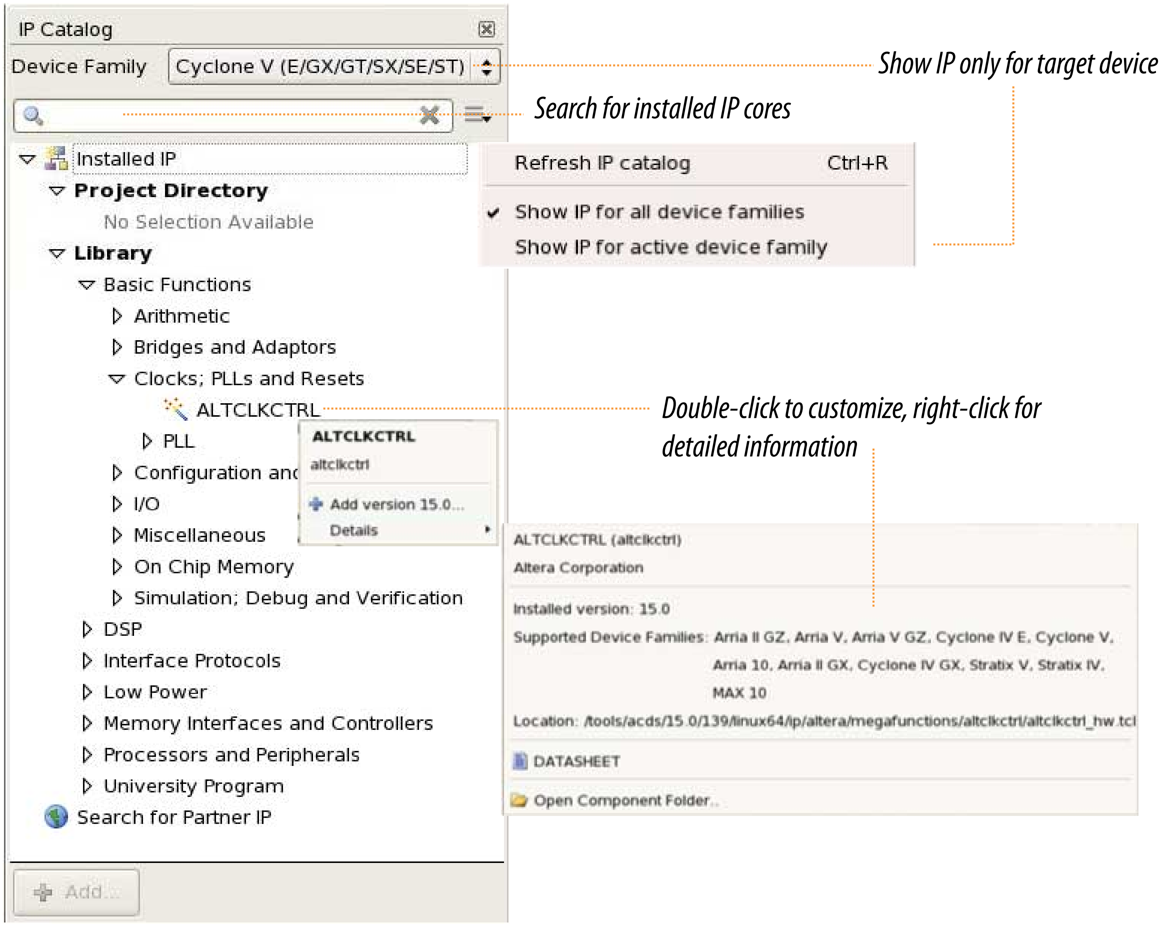
Note: The IP Catalog (**Tools** > **IP Catalog)** and parameter editor replace the MegaWizard™ Plug-In Manager for IP selection and parameterization, beginning in Quartus II software version 14.0. Use the IP Catalog and parameter editor to locate and paramaterize Altera IP cores.

The IP Catalog lists installed IP cores available for your design. Double-click any IP core to launch the parameter editor and generate files representing your IP variation. The parameter editor prompts you to specify an IP variation name, optional ports, and output file generation options. The parameter editor generates a top-level Qsys system file (**.qsys**) or Quartus II IP file (**.qip**) representing the IP core in your project. You can also parameterize an IP variation without an open project.

Use the following features to help you quickly locate and select an IP core:

- Filter IP Catalog to **Show IP for active device family** or **Show IP for all device families**. If you have no project open, select the **Device Family** in IP Catalog.
- Type in the Search field to locate any full or partial IP core name in IP Catalog.
- Right-click an IP core name in IP Catalog to display details about supported devices, open the IP core's installation folder, and view links to documentation.
- Click **Search for Partner IP**, to access partner IP information on the Altera website.

Figure 1-12: Quartus II IP Catalog



IP Catalog

Device Family: Cyclone V (E/GX/GT/SX/SE/ST)

- Installed IP
  - Project Directory
    - No Selection Available
  - Library
    - Basic Functions
      - Arithmetic
      - Bridges and Adaptors
      - Clocks; PLLs and Resets
        - ALTCLKCTRL
        - PLL
      - Configuration and
      - I/O
      - Miscellaneous
      - On Chip Memory
      - Simulation; Debug and Verification
    - DSP
    - Interface Protocols
    - Low Power
    - Memory Interfaces and Controllers
    - Processors and Peripherals
    - University Program
  - Search for Partner IP

ALTCLKCTRL (altclkctrl)
Altera Corporation
Installed version: 15.0
Supported Device Families: Arria II GZ, Arria V, Arria V GZ, Cyclone IV E, Cyclone V, Arria 10, Arria II GX, Cyclone IV GX, Stratix V, Stratix IV, MAX 10
Location: /tools/acds/15.0/139/linux64/ip/altera/megafunctions/altclkctrl/altclkctrl_hw.tcl

Show IP only for target device
Search for installed IP cores
Double-click to customize, right-click for detailed information
Refresh IP catalog Ctrl+R
✓ Show IP for all device families
Show IP for active device family
ALTCLKCTRL
altclkctrl
Add version 15.0...
Details
DATASHEET
Open Component Folder...
Add...

Note: The IP Catalog is also available in Qsys (**View** > **IP Catalog**). The Qsys IP Catalog includes exclusive system interconnect, video and image processing, and other system-level IP that are not available in the Quartus II IP Catalog. For more information about using the Qsys IP Catalog, refer to *Creating a System with Qsys* in the *Quartus II Handbook*.

Related Information

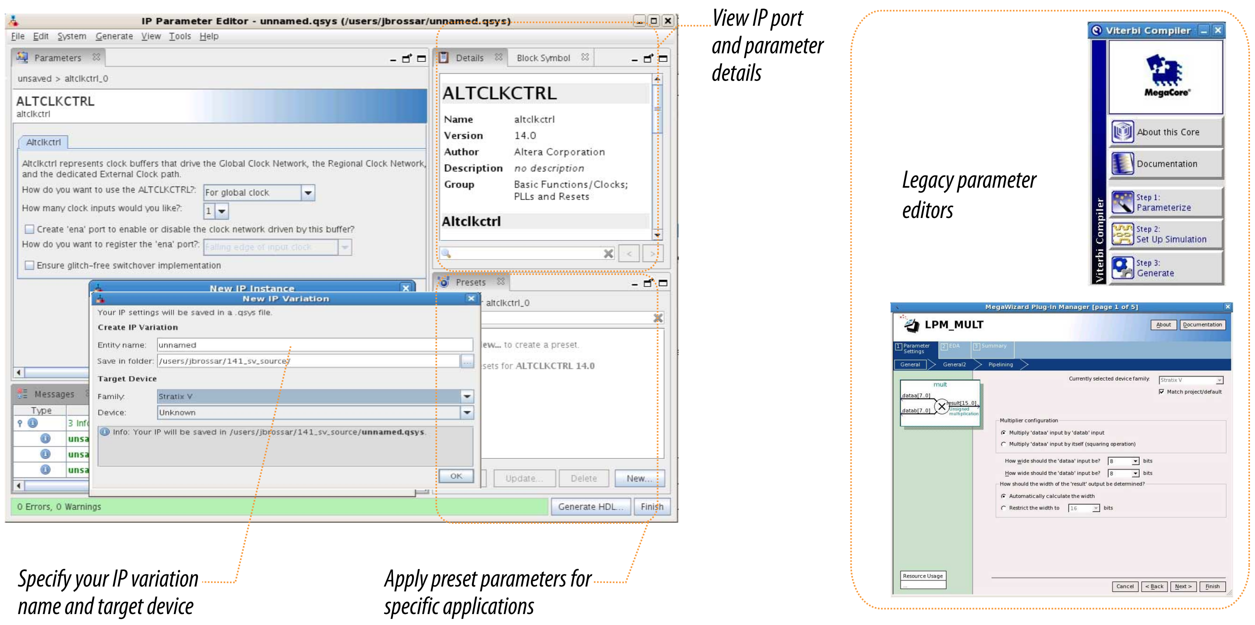
Creating a System With Qsys on page 5-1

Using the Parameter Editor

The parameter editor helps you to configure IP core ports, parameters, and output file generation options.

- Use preset settings in the parameter editor (where provided) to instantly apply preset parameter values for specific applications.
- View port and parameter descriptions, and links to documentation.
- Generate testbench systems or example designs (where provided).

Figure 1-13: IP Parameter Editors


View IP port and parameter details
Legacy parameter editors
Specify your IP variation name and target device
Apply preset parameters for specific applications

Adding IP Cores to IP Catalog

The IP Catalog automatically displays Altera IP cores found in the project directory, in the Altera installation directory, and in the defined IP search path. The IP Catalog can include Altera-provided IP components, third-party IP components, custom IP components that you provide, and previously generated Qsys systems.

You can use the **IP Search Path** option (**Tools** > **Options**) to include custom and third-party IP components in the IP Catalog. The IP Catalog displays all IP cores in the IP search path. The Quartus II software searches the directories listed in the IP search path for the following IP core files:

- Component Description File (**_hw.tcl**)—Defines a single IP core.
- IP Index File (**.ipx**)—Each **.ipx** file indexes a collection of available IP cores, or a reference to other directories to search. In general, **.ipx** files facilitate faster searches.

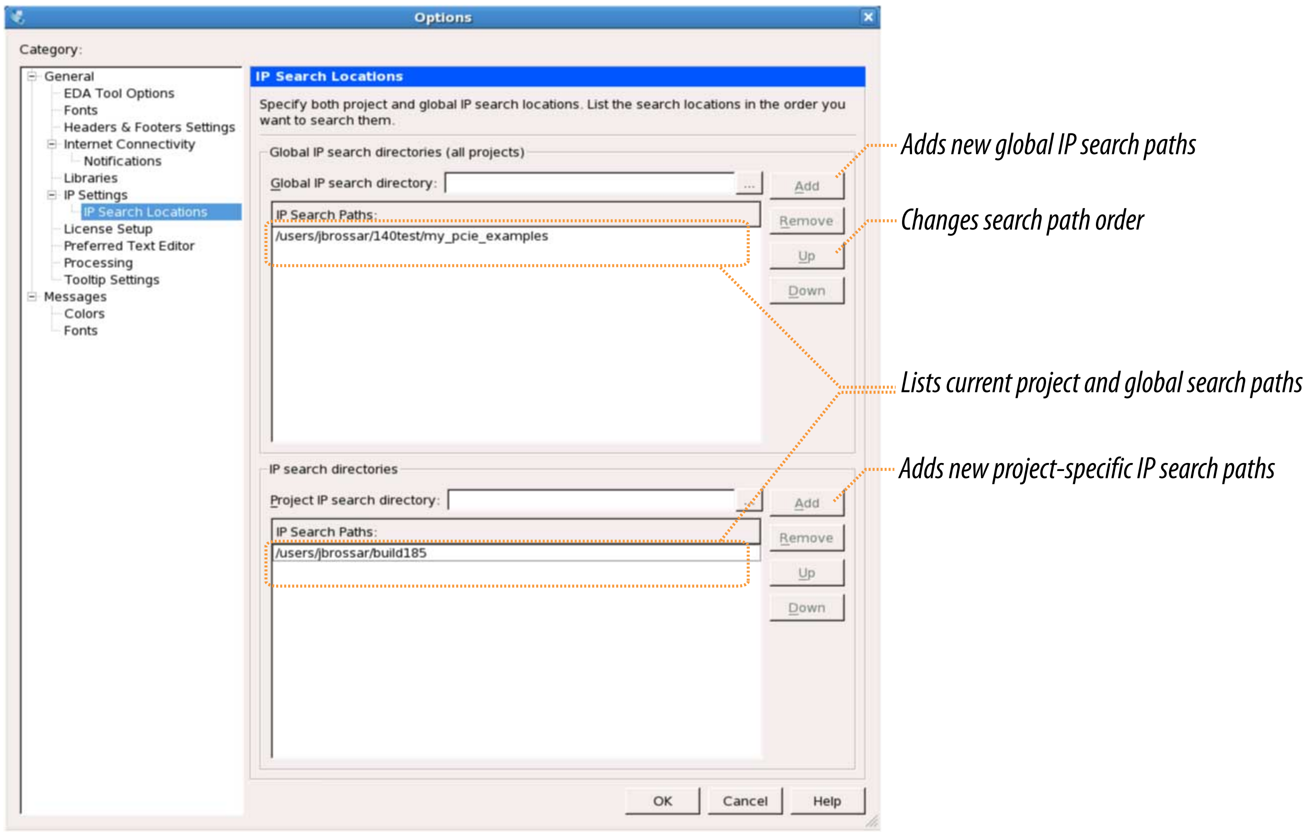
The Quartus II software searches some directories recursively and other directories only to a specific depth. When the search is recursive, the search stops at any directory that contains an **_hw.tcl** or **.ipx** file.

In the following list of search locations, a recursive descent is annotated by **. A single * signifies any file.

Table 1-2: IP Search Locations

Location	Description
PROJECT_DIR/*	Finds IP components and index files in the Quartus II project directory.
PROJECT_DIR/ip//***	Finds IP components and index files in any subdirectory of the **/ip** subdirectory of the Quartus II project directory.

Figure 1-14: Specifying IP Search Locations



IP Search Locations

Specify both project and global IP search locations. List the search locations in the order you want to search them.

Global IP search directories (all projects)

Global IP search directory:

IP Search Paths:
/users/jbrossar/140test/my_pcie_examples

IP search directories

Project IP search directory:

IP Search Paths:
/users/jbrossar/build185

If the Quartus II software recognizes two IP cores with the same name, the following search path precedence rules determine the resolution of files:

1. Project directory.
2. Project database directory.
3. Project IP search path specified in **IP Search Locations**, or with the SEARCH_PATH assignment in the Quartus II Settings File (**.qsf**) for the current project revision.
4. Global IP search path specified in **IP Search Locations**, or with the SEARCH_PATH assignment in the **quartus2.ini** file.
5. Quartus software libraries directory, such as **<*Quartus Installation*>\libraries**.

Note: If you add a component to the search path, you must refresh your system by clicking **File** > **Refresh** to update the IP Catalog.

General Settings for IP

You can use the following settings to control how the Quartus II software manages IP cores in your project.

Table 1-3: IP Core General Setting Locations

Setting Location	Description
Tools > **Options** > **IP Settings** Or **Assignments** > **Settings** > **IP Settings** (only enabled with open project)	• Specify your **IP generation HDL preference**. The parameter editor generates IP files in your preferred HDL by default. • Increase **Maximum Qsys memory usage size** if you experience slow processing for large systems, or if Qsys reports an Out of Memory error. • Specify whether to **Automatically add Quartus II IP files** to all projects. Disable this option to control addition of IP files manually. You may want to experiment with IP before adding to a project. • Use the **IP Regeneration Policy** setting to control when synthesis files are regenerated for each IP variation. Typically you **Always regenerate synthesis files for IP cores** after making changes to an IP variation.
Tools > **Options** > **IP Catalog Search Locations** Or **Assignments** > **Settings** > **IP Catalog Search Locations**	• Specify project and global IP search locations. The Quartus II software searches for IP cores in the project directory, in the Altera installation directory, and in the IP search path.
Assignments > **Settings** > **Simulation**	• **NativeLink Settings** allow you to automatically compile testbenches for supported simulators. You can also specify a script to compile the testbench, and a script to set up the simulation.

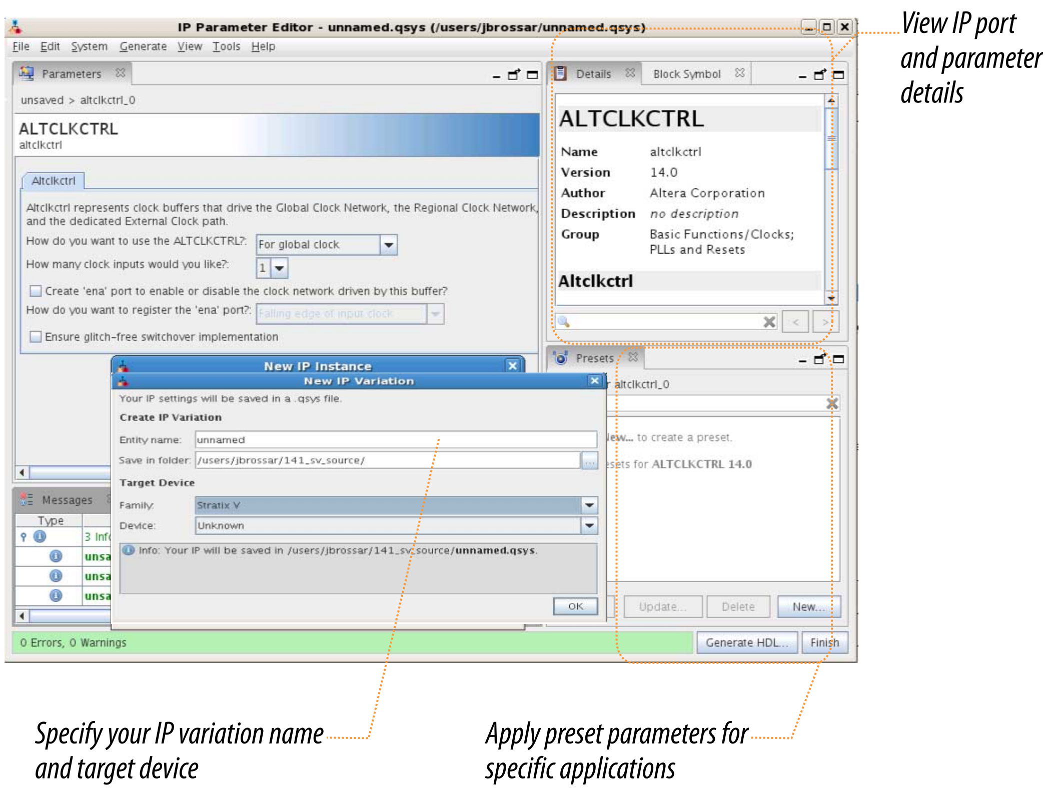
Specifying IP Core Parameters and Options

You can quickly configure a custom IP variation in the parameter editor. Use the following steps to specify IP core options and parameters in the parameter editor. Refer to *Specifying IP Core Parameters and Options (Legacy Parameter Editors)* for configuration of IP cores using the legacy parameter editor.

1. In the IP Catalog (**Tools** > **IP Catalog**), locate and double-click the name of the IP core to customize. The parameter editor appears.
2. Specify a top-level name for your custom IP variation. The parameter editor saves the IP variation settings in a file named <*your_ip*>**.qsys**. Click **OK**.
3. Specify the parameters and options for your IP variation in the parameter editor, including one or more of the following. Refer to your IP core user guide for information about specific IP core parameters.
   - Optionally select preset parameter values if provided for your IP core. Presets specify initial parameter values for specific applications.
   - Specify parameters defining the IP core functionality, port configurations, and device-specific features.
   - Specify options for processing the IP core files in other EDA tools.
4. Click **Generate HDL**, the **Generation** dialog box appears.
5. Specify output file generation options, and then click **Generate**. The IP variation files generate according to your specifications.

6. To generate a simulation testbench, click **Generate** > **Generate Testbench System**.
7. To generate an HDL instantiation template that you can copy and paste into your text editor, click **Generate** > **HDL Example**.
8. Click **Finish**. The parameter editor adds the top-level **.qsys** file to the current project automatically. If you are prompted to manually add the **.qsys** file to the project, click **Project** > **Add/Remove Files in Project** to add the file.
9. After generating and instantiating your IP variation, make appropriate pin assignments to connect ports.

Figure 1-15: IP Parameter Editor


View IP port and parameter details
Specify your IP variation name and target device
Apply preset parameters for specific applications
IP Parameter Editor - unnamed.qsys (/users/jbrossar/unnamed.qsys)
File Edit System Generate View Tools Help
Parameters
Details
Block Symbol
Presets
Messages
New IP Instance
New IP Variation
OK
Update...
Delete
New...
Generate HDL...
Finish

unsaved > altclkctrl_0

ALTCLKCTRL
altclkctrl

Altclkctrl represents clock buffers that drive the Global Clock Network, the Regional Clock Network, and the dedicated External Clock path.

How do you want to use the ALTCLKCTRL?: For global clock
How many clock inputs would you like?: 1
☐ Create 'ena' port to enable or disable the clock network driven by this buffer?
How do you want to register the 'ena' port?: Falling edge of input clock
☐ Ensure glitch-free switchover implementation

ALTCLKCTRL

Name	altclkctrl
Version	14.0
Author	Altera Corporation
Description	no description
Group	Basic Functions/Clocks; PLLs and Resets

Altclkctrl

Your IP settings will be saved in a .qsys file.

Create IP Variation
Entity name: unnamed
Save in folder: /users/jbrossar/141_sv_source/

Target Device
Family: Stratix V
Device: Unknown

Info: Your IP will be saved in /users/jbrossar/141_sv_source/unnamed.qsys.

altclkctrl_0
...ew... to create a preset.
...sets for ALTCLKCTRL 14.0

0 Errors, 0 Warnings

Files Generated for Altera IP Cores

The Quartus II software generates the following IP core output file structure:

Figure 1-16: IP Core Generated Files

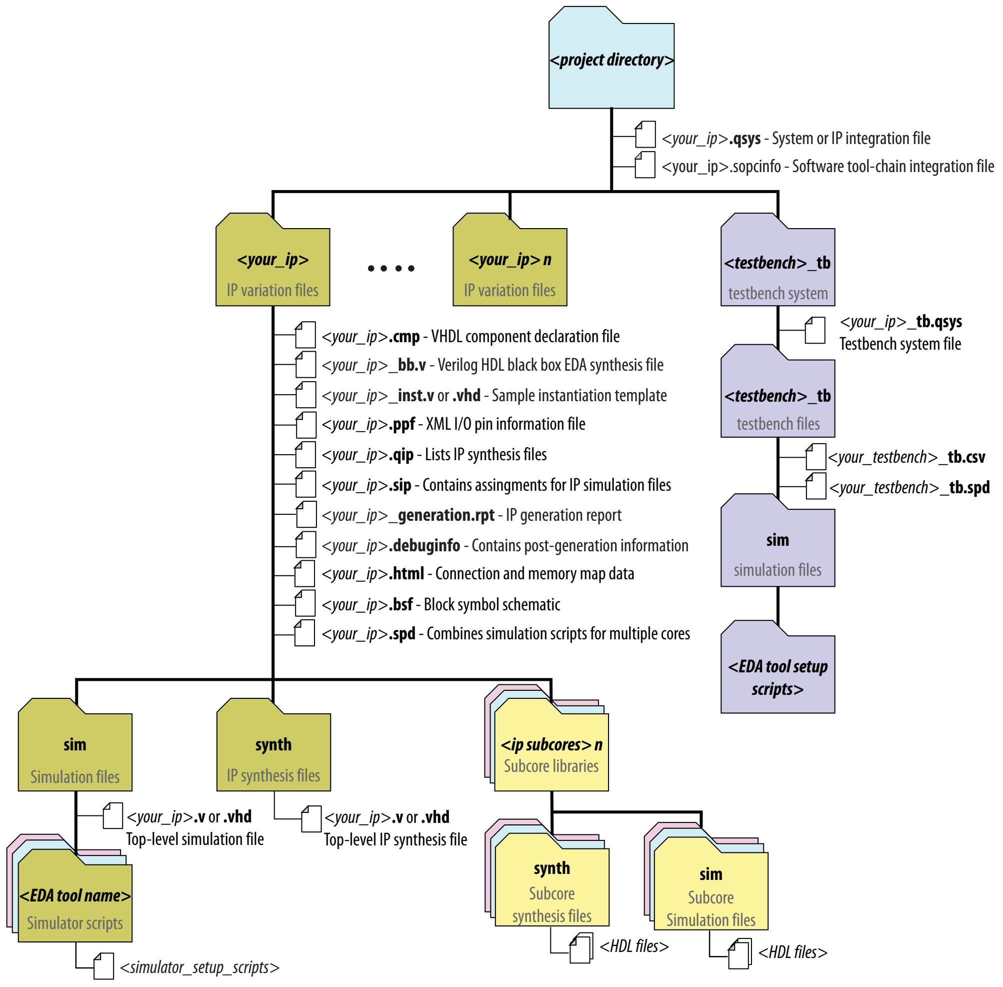

<project directory>
<your_ip>.qsys - System or IP integration file
<your_ip>.sopcinfo - Software tool-chain integration file
<your_ip> IP variation files
<your_ip> n IP variation files
<testbench>_tb testbench system
<your_ip>_tb.qsys Testbench system file
<testbench>_tb testbench files
<your_testbench>_tb.csv
<your_testbench>_tb.spd
sim simulation files
<EDA tool setup scripts>
<your_ip>.cmp - VHDL component declaration file
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_inst.v or .vhd - Sample instantiation template
<your_ip>.ppf - XML I/O pin information file
<your_ip>.qip - Lists IP synthesis files
<your_ip>.sip - Contains assingments for IP simulation files
<your_ip>_generation.rpt - IP generation report
<your_ip>.debuginfo - Contains post-generation information
<your_ip>.html - Connection and memory map data
<your_ip>.bsf - Block symbol schematic
<your_ip>.spd - Combines simulation scripts for multiple cores
sim Simulation files
synth IP synthesis files
<ip subcores> n Subcore libraries
<your_ip>.v or .vhd Top-level simulation file
<your_ip>.v or .vhd Top-level IP synthesis file
<EDA tool name> Simulator scripts
<simulator_setup_scripts>
synth Subcore synthesis files
sim Subcore Simulation files
<HDL files>
<HDL files>

Table 1-4: IP Core Generated Files

File Name	Description
<*my_ip*>.qsys	The Qsys system or top-level IP variation file. <*my_ip*> is the name that you give your IP variation.
<*system*>.sopcinfo	Describes the connections and IP component parameterizations in your Qsys system. You can parse its contents to get requirements when you develop software drivers for IP components. Downstream tools such as the Nios II tool chain use this file. The **.sopcinfo** file and the **system.h** file generated for the Nios II tool chain include address map information for each slave relative to each master that accesses the slave. Different masters may have a different address map to access a particular slave component.

File Name	Description
<*my_ip*>.cmp	The VHDL Component Declaration (**.cmp**) file is a text file that contains local generic and port definitions that you can use in VHDL design files.
<*my_ip*>.html	A report that contains connection information, a memory map showing the address of each slave with respect to each master to which it is connected, and parameter assignments.
<*my_ip*>_generation.rpt	IP or Qsys generation log file. A summary of the messages during IP generation.
<*my_ip*>.debuginfo	Contains post-generation information. Used to pass System Console and Bus Analyzer Toolkit information about the Qsys interconnect. The Bus Analysis Toolkit uses this file to identify debug components in the Qsys interconnect.
<*my_ip*>.qip	Contains all the required information about the IP component to integrate and compile the IP component in the Quartus II software.
<*my_ip*>.csv	Contains information about the upgrade status of the IP component.
<*my_ip*>.bsf	A Block Symbol File (**.bsf**) representation of the IP variation for use in Quartus II Block Diagram Files (**.bdf**).
<*my_ip*>.spd	Required input file for `ip-make-simscript` to generate simulation scripts for supported simulators. The **.spd** file contains a list of files generated for simulation, along with information about memories that you can initialize.
<*my_ip*>.ppf	The Pin Planner File (**.ppf**) stores the port and node assignments for IP components created for use with the Pin Planner.
<*my_ip*>_bb.v	You can use the Verilog black-box (**_bb.v**) file as an empty module declaration for use as a black box.
<*my_ip*>.sip	Contains information required for NativeLink simulation of IP components. You must add the **.sip** file to your Quartus project.
<*my_ip*>_inst.v or **_inst.vhd**	HDL example instantiation template. You can copy and paste the contents of this file into your HDL file to instantiate the IP variation.
<*my_ip*>.regmap	If the IP contains register information, the **.regmap** file generates. The **.regmap** file describes the register map information of master and slave interfaces. This file complements the **.sopcinfo** file by providing more detailed register information about the system. This enables register display views and user customizable statistics in System Console.

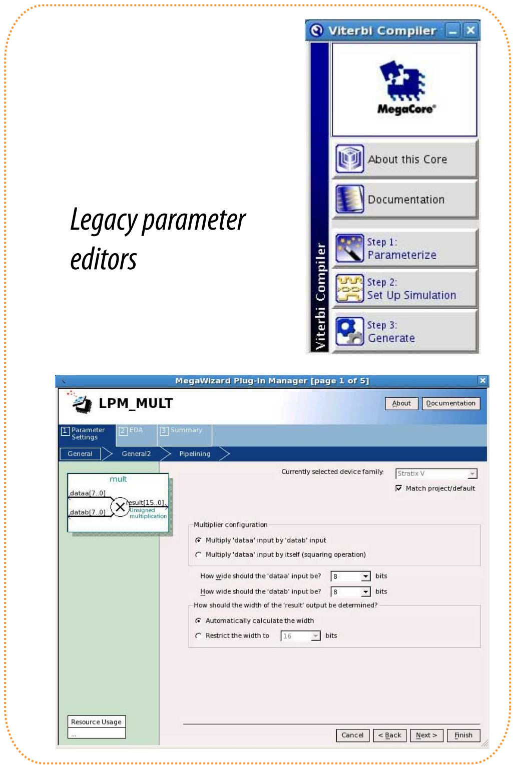
File Name	Description
***<my_ip>*.svd**	Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Qsys system. During synthesis, the **.svd** files for slave interfaces visible to System Console masters are stored in the **.sof** file in the debug section. System Console reads this section, which Qsys can query for register map information. For system slaves, Qsys can access the registers by name.
***<my_ip>*.v** or ***<my_ip>*.vhd**	HDL files that instantiate each submodule or child IP core for synthesis or simulation.
mentor/	Contains a ModelSim® script **msim_setup.tcl** to set up and run a simulation.
aldec/	Contains a Riviera-PRO script **rivierapro_setup.tcl** to setup and run a simulation.
/synopsys/vcs **/synopsys/vcsmx**	Contains a shell script **vcs_setup.sh** to set up and run a VCS® simulation. Contains a shell script **vcsmx_setup.sh** and **synopsys_ sim.setup** file to set up and run a VCS MX® simulation.
/cadence	Contains a shell script **ncsim_setup.sh** and other setup files to set up and run an NCSIM simulation.
/submodules	Contains HDL files for the IP core submodule.
***<child IP cores>*/**	For each generated child IP core directory, Qsys generates **/synth** and **/ sim** sub-directories.

Specifying IP Core Parameters and Options (Legacy Parameter Editors)

Some IP cores use a legacy version of the parameter editor for configuration and generation. Use the following steps to configure and generate an IP variation using a legacy parameter editor.

Note: The legacy parameter editor generates a different output file structure than the latest parameter editor. Refer to *Specifying IP Core Parameters and Options* for configuration of IP cores that use the latest parameter editor.

Figure 1-17: Legacy Parameter Editors



Currently selected device family: Stratix V
[✓] Match project/default

Multiplier configuration
[✓ Multiply 'dataa' input by 'datab' input]
Multiply 'dataa' input by itself (squaring operation)

How wide should the 'dataa' input be? 8 bits
How wide should the 'datab' input be? 8 bits

How should the width of the 'result' output be determined?
[✓ Automatically calculate the width]
Restrict the width to 16 bits

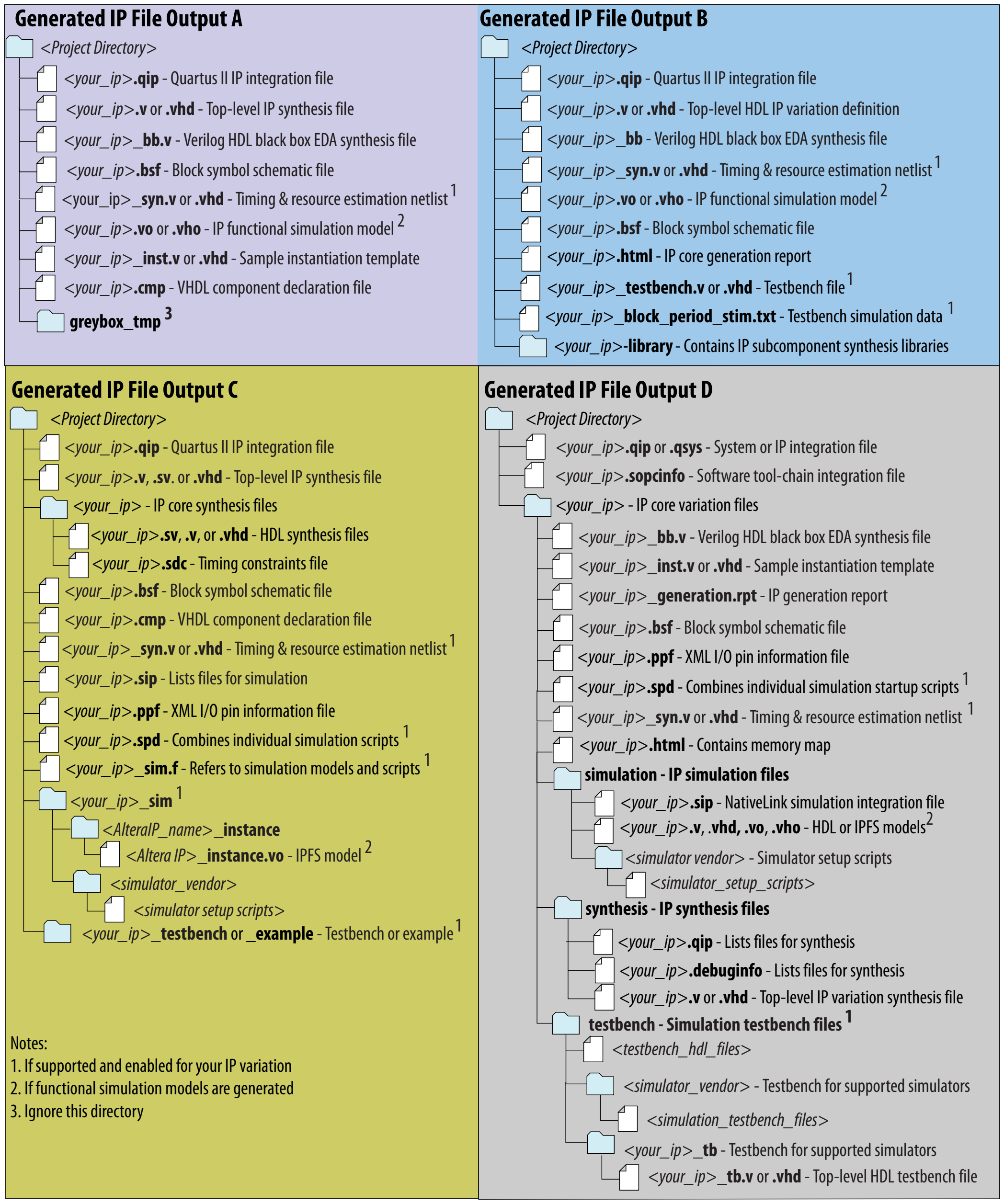
1. In the IP Catalog (**Tools** > **IP Catalog**), locate and double-click the name of the IP core to customize. The parameter editor appears.
2. Specify a top-level name and output HDL file type for your IP variation. This name identifies the IP core variation files in your project. Click **OK**.
3. Specify the parameters and options for your IP variation in the parameter editor. Refer to your IP core user guide for information about specific IP core parameters.
4. Click **Finish** or **Generate** (depending on the parameter editor version). The parameter editor generates the files for your IP variation according to your specifications. Click **Exit** if prompted when generation is complete. The parameter editor adds the top-level **.qip** file to the current project automatically.

   Note: To manually add an IP variation generated with legacy parameter editor to a project, click **Project** > **Add/Remove Files in Project** and add the IP variation **.qip** file.

Files Generated for Altera IP Cores (Legacy Parameter Editors)

The Quartus II software generates one of the following output file structures for Altera IP cores that use a legacy parameter editor.

Figure 1-18: IP Core Generated Files (Legacy Parameter Editor)


Generated IP File Output A
<Project Directory>
<your_ip>.qip - Quartus II IP integration file
<your_ip>.v or .vhd - Top-level IP synthesis file
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>.bsf - Block symbol schematic file
<your_ip>_syn.v or .vhd - Timing & resource estimation netlist 1
<your_ip>.vo or .vho - IP functional simulation model 2
<your_ip>_inst.v or .vhd - Sample instantiation template
<your_ip>.cmp - VHDL component declaration file
greybox_tmp 3
Generated IP File Output B
<Project Directory>
<your_ip>.qip - Quartus II IP integration file
<your_ip>.v or .vhd - Top-level HDL IP variation definition
<your_ip>_bb - Verilog HDL black box EDA synthesis file
<your_ip>_syn.v or .vhd - Timing & resource estimation netlist 1
<your_ip>.vo or .vho - IP functional simulation model 2
<your_ip>.bsf - Block symbol schematic file
<your_ip>.html - IP core generation report
<your_ip>_testbench.v or .vhd - Testbench file 1
<your_ip>_block_period_stim.txt - Testbench simulation data 1
<your_ip>-library - Contains IP subcomponent synthesis libraries
Generated IP File Output C
<Project Directory>
<your_ip>.qip - Quartus II IP integration file
<your_ip>.v, .sv. or .vhd - Top-level IP synthesis file
<your_ip> - IP core synthesis files
<your_ip>.sv, .v, or .vhd - HDL synthesis files
<your_ip>.sdc - Timing constraints file
<your_ip>.bsf - Block symbol schematic file
<your_ip>.cmp - VHDL component declaration file
<your_ip>_syn.v or .vhd - Timing & resource estimation netlist 1
<your_ip>.sip - Lists files for simulation
<your_ip>.ppf - XML I/O pin information file
<your_ip>.spd - Combines individual simulation scripts 1
<your_ip>_sim.f - Refers to simulation models and scripts 1
<your_ip>_sim 1
<AlteraIP_name>_instance
<Altera IP>_instance.vo - IPFS model 2
<simulator_vendor>
<simulator setup scripts>
<your_ip>_testbench or _example - Testbench or example 1
Notes:
1. If supported and enabled for your IP variation
2. If functional simulation models are generated
3. Ignore this directory
Generated IP File Output D
<Project Directory>
<your_ip>.qip or .qsys - System or IP integration file
<your_ip>.sopcinfo - Software tool-chain integration file
<your_ip> - IP core variation files
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_inst.v or .vhd - Sample instantiation template
<your_ip>_generation.rpt - IP generation report
<your_ip>.bsf - Block symbol schematic file
<your_ip>.ppf - XML I/O pin information file
<your_ip>.spd - Combines individual simulation startup scripts 1
<your_ip>_syn.v or .vhd - Timing & resource estimation netlist 1
<your_ip>.html - Contains memory map
simulation - IP simulation files
<your_ip>.sip - NativeLink simulation integration file
<your_ip>.v, .vhd, .vo, .vho - HDL or IPFS models 2
<simulator vendor> - Simulator setup scripts
<simulator_setup_scripts>
synthesis - IP synthesis files
<your_ip>.qip - Lists files for synthesis
<your_ip>.debuginfo - Lists files for synthesis
<your_ip>.v or .vhd - Top-level IP variation synthesis file
testbench - Simulation testbench files 1
<testbench_hdl_files>
<simulator_vendor> - Testbench for supported simulators
<simulation_testbench_files>
<your_ip>_tb - Testbench for supported simulators
<your_ip>_tb.v or .vhd - Top-level HDL testbench file

Note: To manually add an IP variation to a Quartus II project, click **Project** > **Add/Remove Files in Project** and add only the IP variation **.qip** or **.qsys** file, but not both, to the project. Do not manually add the top-level HDL file to the project.

Scripting IP Core Generation

You can alternatively use command-line utilities to define and generate an IP core variation outside of the Quartus II GUI.Use `qsys-script` to run a Tcl file that parameterizes an IP variation you define in a script. Then, use `qsys-generate` to generate a **.qsys** file representing your parameterized IP variation.

The `qsys-generate` command is the same as when generating using the Qsys GUI. For command-line help listing all options for these executables, type <*executable name*> `--help`

To create a instance of a parameterizable Altera IP core at the command line, rather than using the GUI, follow these steps:

1. Run `qsys-script` to execute a Tcl script, similar to the example, that instantiates the IP and sets the IP parameters defined by the script:

```
qsys-script --script=<script_file>.tcl
```

2. Run `qsys-generate` to generate the RTL for the IP variation:

```
qsys-generate <IP variation file>.qsys
```

Note: Creating an IP generation script is an advanced feature that requires access to special IP core parameters. For more information about creating an IP generation script, contact your Altera sales representative.

Table 1-5: qsys-generate Command-Line Options

Option	Usage	Description
`<1st arg file>`	Required	The name of the **.qsys** system file to generate.
`--synthesis=<VERILOG|VHDL>`	Optional	Creates synthesis HDL files that Qsys uses to compile the system in a Quartus II project. You must specify the preferred generation language for the top-level RTL file for the generated Qsys system.
`--block-symbol-file`	Optional	Creates a Block Symbol File (**.bsf**) for the Qsys system.
`--simulation=<VERILOG|VHDL>`	Optional	Creates a simulation model for the Qsys system. The simulation model contains generated HDL files for the simulator, and may include simulation-only features. You must specify the preferred simulation language.
`--testbench=<SIMPLE|STANDARD>`	Optional	Creates a testbench system that instantiates the original system, adding bus functional models (BFMs) to drive the top-level interfaces. When you generate the system, the BFMs interact with the system in the simulator.
`--testbench-simulation=<VERILOG|VHDL>`	Optional	After you create the testbench system, you can create a simulation model for the testbench system.

Option	Usage	Description
`--search-path=<value>`	Optional	If you omit this command, Qsys uses a standard default path. If you provide this command, Qsys searches a comma-separated list of paths. To include the standard path in your replacement, use "`$`", for example, "`/extra/dir,$`".
`--jvm-max-heap-size=<value>`	Optional	The maximum memory size that Qsys uses for allocations when running `qsys-generate`. You specify the value as `<size><unit>`, where `unit` is m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512m.
`--family=<value>`	Optional	Specifies the device family.
`--part=<value>`	Optional	Specifies the device part number. If set, this option overrides the `--family` option.
`--allow-mixed-language-simulation`	Optional	Enables a mixed language simulation model generation. If true, if a preferred simulation language is set, Qsys uses a `fileset` of the component for the simulation model generation. When false, which is the default, Qsys uses the language specified with `--file-set=<value>` for all components for simulation model generation. The current version of the ModelSim-Altera simulator supports mixed language simulation.

Modifying an IP Variation

You can easily modify the parameters of any Altera IP core variation in the parameter editor to match your design requirements. Use any of the following methods to modify an IP variation in the parameter editor.

Table 1-6: Modifying an IP Variation

Menu Command	Action
File > **Open**	Select the top-level HDL (**.v**, or **.vhd**) IP variation file to launch the parameter editor and modify the IP variation. Regenerate the IP variation to implement your changes.
View > **Utility Windows** > **Project Navigator** > **IP Components**	Double-click the IP variation to launch the parameter editor and modify the IP variation. Regenerate the IP variation to implement your changes.

Menu Command	Action
Project > **Upgrade IP Components**	Select the IP variation and click **Upgrade in Editor** to launch the parameter editor and modify the IP variation. Regenerate the IP variation to implement your changes.

Upgrading IP Cores

IP core variants generated with a previous version of the Quartus II software may require upgrading before use in the current version of the Quartus II software. Click **Project** > **Upgrade IP Components** to identify and upgrade outdated IP core variants.

Icons in the **Upgrade IP Components** dialog box indicate when IP upgrade is required, optional, or unsupported for IP cores in your design. This dialog box may open automatically when you open a project containing upgradeable IP variations. You must upgrade IP cores that require upgrade before you can compile the IP variation in the current version of the Quartus II software.

The upgrade process preserves the original IP variation file in the project directory as <*my_variant*>_**BAK.qsys** for IP targeting Arria 10 and later devices, and as <*my_variant*>**_BAK.v**, **.sv**, or **.vhd** for legacy IP targeting 28nm devices and greater.

Note: Upgrading IP cores for Arria 10 and later devices may append a unique identifier to the original IP core entity name(s), without similarly modifying the IP instance name. There is no requirement to update these entity references in any supporting Quartus II file; such as the Quartus II Settings File (**.qsf**), Synopsys Design Constraints File (**.sdc**), or SignalTap File (**.stp**), if these files contain instance names. The Quartus II software reads only the instance name and ignores the entity name in paths that specify both names. Use only instance names in assignments.

Table 1-7: IP Core Upgrade Status

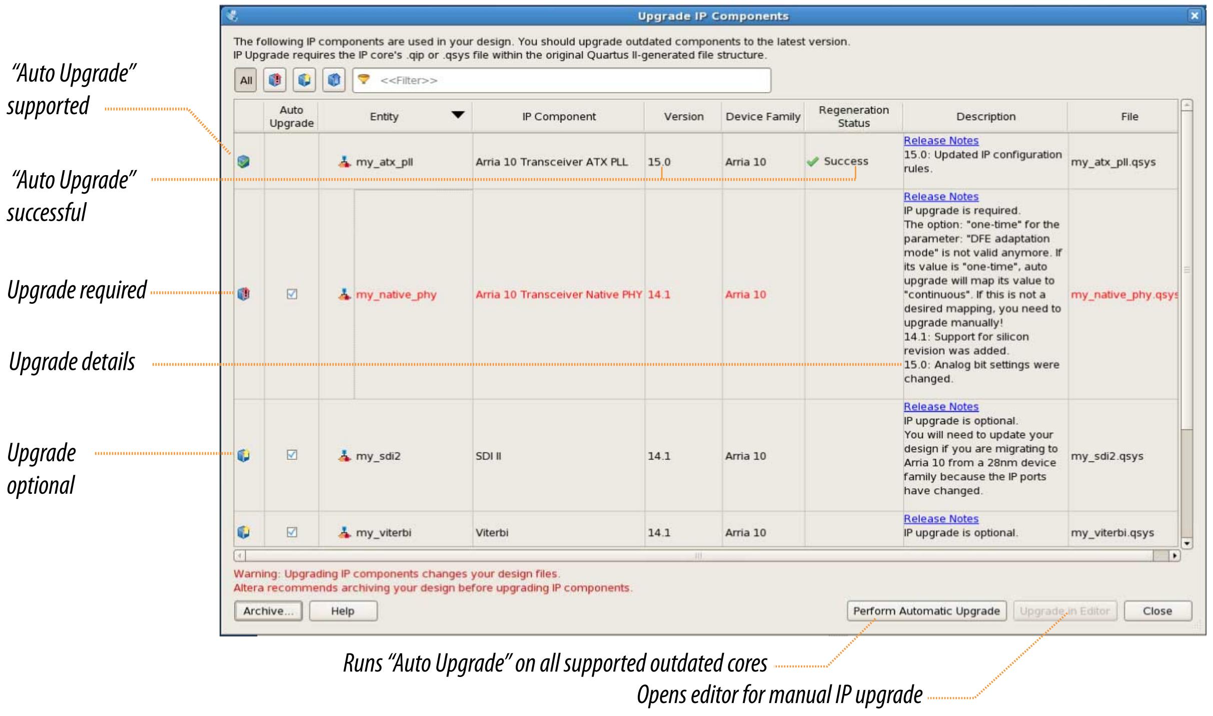
IP Core Status	Description
IP Upgraded ✓	Your IP variation uses the lastest version of the IP core.
IP Upgrade Optional ↑	Upgrade is optional for this IP variation in the current version of the Quartus II software. You can upgrade this IP variation to take advantage of the latest development of this IP core. Alternatively you can retain previous IP core characteristics by declining to upgrade. Refer to the Description for details about IP core version differences. If you do not upgrade the IP, the IP variation synthesis and simulation files are unchanged and you cannot modify parameters until upgrading.
IP Upgrade Mismatch Warning !	Warning of non-critical IP core differences in migrating IP to another device family.

IP Core Status	Description
IP Upgrade Required	You must upgrade the IP variation before compiling in the current version of the Quartus II software. Refer to the Description for details about IP core version differences.
IP Upgrade Unspported	Upgrade of the IP variation is not supported in the current version of the Quartus II software due to incompatibility with the current version of the Quartus II software. You are prompted to replace the unsupported IP core with a supported equivalent IP core from the IP Catalog. Refer to the Description for details about IP core version differences and links to Release Notes.
IP End of Life	Altera designates the IP core as end-of-life status. You may or may not be able to edit the IP core in the parameter editor. Support for this IP core discontinues in future releases of the Quartus II software.
Encrypted IP Core	The IP variation is encrypted.

Follow these steps to upgrade IP cores:

1. In the latest version of the Quartus II software, open the Quartus II project containing an outdated IP core variation. The **Upgrade IP Components** dialog automatically displays the status of IP cores in your project, along with instructions for upgrading each core. Click **Project** > **Upgrade IP Components** to access this dialog box manually.
2. To upgrade one or more IP cores that support automatic upgrade, ensure that the **Auto Upgrade** option is turned on for the IP core(s), and then click **Perform Automatic Upgrade**. The **Status** and **Version** columns update when upgrade is complete. Example designs provided with any Altera IP core regenerate automatically whenever you upgrade an IP core.
3. To manually upgrade an individual IP core, select the IP core and then click **Upgrade in Editor** (or simply double-click the IP core name. The parameter editor opens, allowing you to adjust parameters and regenerate the latest version of the IP core.

Figure 1-19: Upgrading IP Cores



The following IP components are used in your design. You should upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original Quartus II-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description	File
		my_atx_pll	Arria 10 Transceiver ATX PLL	15.0	Arria 10	✓ Success	Release Notes 15.0: Updated IP configuration rules.	my_atx_pll.qsys
	☑	my_native_phy	Arria 10 Transceiver Native PHY	14.1	Arria 10		Release Notes IP upgrade is required. The option: "one-time" for the parameter: "DFE adaptation mode" is not valid anymore. If its value is "one-time", auto upgrade will map its value to "continuous". If this is not a desired mapping, you need to upgrade manually! 14.1: Support for silicon revision was added. 15.0: Analog bit settings were changed.	my_native_phy.qsys
	☑	my_sdi2	SDI II	14.1	Arria 10		Release Notes IP upgrade is optional. You will need to update your design if you are migrating to Arria 10 from a 28nm device family because the IP ports have changed.	my_sdi2.qsys
	☑	my_viterbi	Viterbi	14.1	Arria 10		Release Notes IP upgrade is optional.	my_viterbi.qsys

Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design before upgrading IP components.

Note: IP cores older than Quartus II software version 12.0 do not support upgrade. Altera verifies that the current version of the Quartus II software compiles the previous version of each IP core. The *Altera IP Release Notes* reports any verification exceptions for Altera IP cores. Altera does not verify compilation for IP cores older than the previous two releases.

Related Information

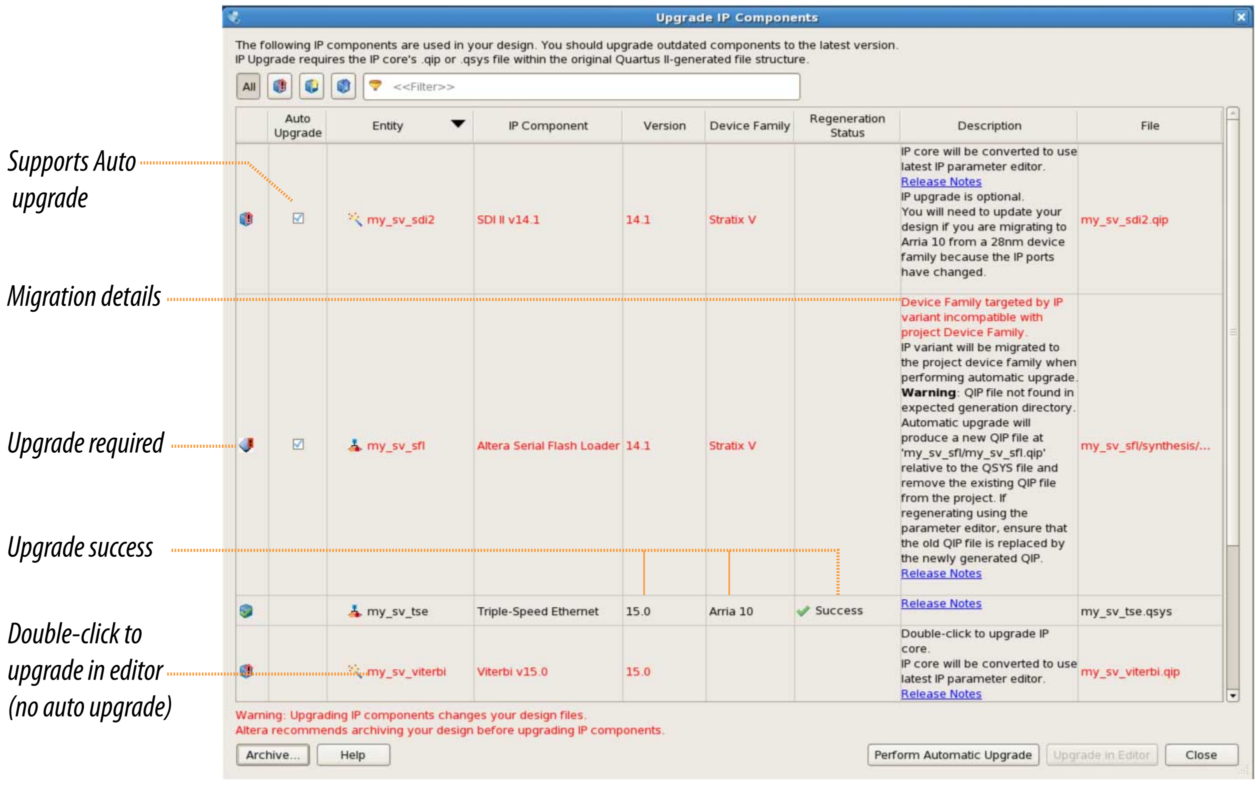
Altera IP Release Notes

Migrating IP Cores to a Different Device

IP migration allows you to target the latest device families with IP originally generated for a different device. Some Altera IP cores migrate automatically, some IP cores require manual IP regeneration, and some do not support device migration and must be replaced in your design.
The text and icons in the **Upgrade IP Components** dialog box identifies the migration support for each IP core in the design.

Note: Migration of some IP cores requires installed support for the original and migration device families. For example, migration from a Stratix V device to an Arria 10 device requires installation of Stratix V and Arria 10 device families with the Quartus II software.

Figure 1-20: Upgrading IP Cores



The following IP components are used in your design. You should upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original Quartus II-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description	File
	☑	my_sv_sdi2	SDI II v14.1	14.1	Stratix V		IP core will be converted to use latest IP parameter editor. Release Notes IP upgrade is optional. You will need to update your design if you are migrating to Arria 10 from a 28nm device family because the IP ports have changed.	my_sv_sdi2.qip
	☑	my_sv_sfl	Altera Serial Flash Loader	14.1	Stratix V		Device Family targeted by IP variant incompatible with project Device Family. IP variant will be migrated to the project device family when performing automatic upgrade. **Warning**: QIP file not found in expected generation directory. Automatic upgrade will produce a new QIP file at 'my_sv_sfl/my_sv_sfl.qip' relative to the QSYS file and remove the existing QIP file from the project. If regenerating using the parameter editor, ensure that the old QIP file is replaced by the newly generated QIP. Release Notes	my_sv_sfl/synthesis/...
		my_sv_tse	Triple-Speed Ethernet	15.0	Arria 10	✓ Success	Release Notes	my_sv_tse.qsys
		my_sv_viterbi	Viterbi v15.0	15.0			Double-click to upgrade IP core. IP core will be converted to use latest IP parameter editor. Release Notes	my_sv_viterbi.qip

Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design before upgrading IP components.

1. Click **File** > **Open Project** and open the Quartus II project containing IP for migration to another device in the original version of the Quartus II software.
2. To specify a different target device for migration, click **Assignments** > **Device** and select the target device family.
3. To display IP cores requiring migration, click **Project** > **Upgrade IP Components**. The **Description** field prompts you to run auto update or double-click IP cores for migration.
4. To migrate one or more IP cores that support automatic upgrade, ensure that the **Auto Upgrade** option is turned on for the IP core(s), and then click **Perform Automatic Upgrade**. The **Status** and **Version** columns update when upgrade is complete.
5. To migrate an IP core that does not support automatic upgrade, double-click the IP core name, and then click **OK**. The parameter editor appears.
   a. If the parameter editor specifies a **Currently selected device family**, turn off **Match project/ default**, and then select the new target device family.
   b. Click **Finish** to migrate the IP variation using best-effort mapping to new parameters and settings. A new parameter editor opens displaying best-effort mapped parameters.
   c. Click **Generate HDL**, and then confirm the **Synthesis** and **Simulation** file options. Verilog HDL is the defauilt output file format specified. If your original IP core was generated for VHDL, select **VHDL** to retain the original output format.

6. To regenerate the new IP variation for the new target device, click **Generate**. When generation is complete, click **Close**.
7. Click **Finish** to complete migration of the IP core. Click **OK** if you are prompted to overwrite IP core files. The **Device Family** column displays the new target device name when migration is complete. The migration process replaces <*my_ip*>**.qip** with the <*my_ip*>**.qsys** top-level IP file in your project.

   Note: If migration does not replace <*my_ip*>**.qip** with <*my_ip*>**.qsys**, click **Project > Add/Remove Files in Project** to replace the file in your project.
8. Review the latest parameters in the parameter editor or generated HDL for correctness. IP migration may change ports, parameters, or functionality of the IP core. During migration, the IP core's HDL generates into a library that is different from the original output location of the IP core. Update any assignments that reference outdated locations. If your upgraded IP core is represented by a symbol in a supporting Block Design File schematic, replace the symbol with the newly generated <*my_ip*>**.bsf** after migration.

   Note: The migration process may change the IP variation interface, parameters, and functionality. This may require you to change your design or to re-parameterize your variant after the **Upgrade IP Components** dialog box indicates that migration is complete. The **Description** field identifies IP cores that require design or parameter changes.

Related Information

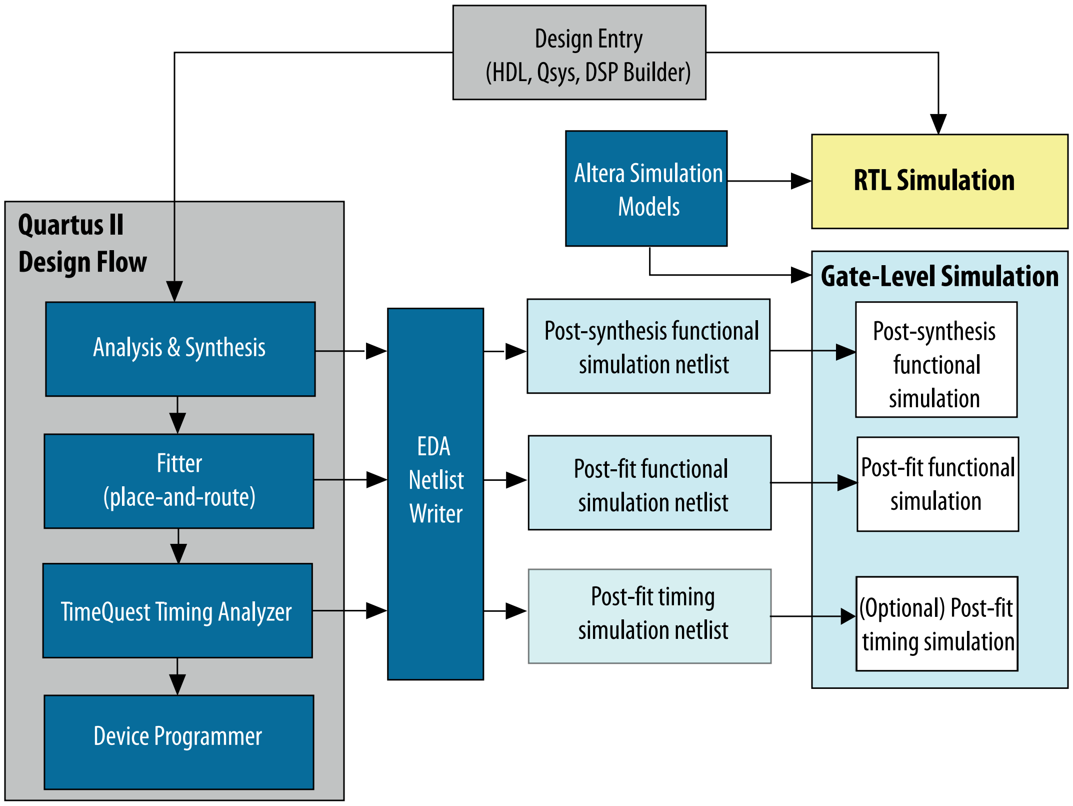
Altera IP Release Notes

Simulating Altera IP Cores in other EDA Tools

The Quartus II software supports RTL and gate-level design simulation of Altera IP cores in supported EDA simulators. Simulation involves setting up your simulator working environment, compiling simulation model libraries, and running your simulation.

You can use the functional simulation model and the testbench or example design generated with your IP core for simulation. The functional simulation model and testbench files are generated in a project subdirectory. This directory may also include scripts to compile and run the testbench. For a complete list of models or libraries required to simulate your IP core, refer to the scripts generated with the testbench. You can use the Quartus II NativeLink feature to automatically generate simulation files and scripts. NativeLink launches your preferred simulator from within the Quartus II software.

Figure 1-21: Simulation in Quartus II Design Flow


Design Entry
(HDL, Qsys, DSP Builder)
Altera Simulation Models
RTL Simulation
Quartus II Design Flow
Analysis & Synthesis
Fitter (place-and-route)
TimeQuest Timing Analyzer
Device Programmer
EDA Netlist Writer
Post-synthesis functional simulation netlist
Post-fit functional simulation netlist
Post-fit timing simulation netlist
Gate-Level Simulation
Post-synthesis functional simulation
Post-fit functional simulation
(Optional) Post-fit timing simulation

Note: Post-fit timing simulation is supported only for Stratix IV and Cyclone IV devices in the current version of the Quartus II software. Altera IP supports a variety of simulation models, including simulation-specific IP functional simulation models and encrypted RTL models, and plain text RTL models. These are all cycle-accurate models. The models support fast functional simulation of your IP core instance using industry-standard VHDL or Verilog HDL simulators. For some cores, only the plain text RTL model is generated, and you can simulate that model. Use the simulation models only for simulation and not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

Related Information

Simulating Altera Designs

Generating Simulation Scripts

You can automatically generate simulation scripts to set up supported simulators. These scripts compile the required device libraries and system design files in the correct order, and then elaborate or load the top-level design for simulation. You can also use scripts to modify the top-level simulation environment, independent of IP simulation files that are replaced during regeneration. You can modify the scripts to set up supported simulators.

Use the NativeLink feature to generate simulation scripts to automate simulation steps. You can reuse these generated files and simulation scripts in a custom simulation flow. NativeLink optionally generates scripts for your simulator in the project subdirectory.

1. Click **Assignments** > **Settings**.
2. Under **EDA Tool Settings,** click **Simulation**.
3. Select the **Tool name** of your simulator.
4. Click **More NativeLink Settings**.
5. Turn on **Generate third-party EDA tool command scripts without running the EDA tool**.

Table 1-8: NativeLink Generated Scripts for RTL Simulation

Simulator(s)	Simulation File	Use
Mentor Graphics ModelSim QuestaSim	**/simulation/modelsim/<*my_ip*>.do**	Source directly with your simulator.
Aldec Riviera Pro	**/simulation/modelsim/<*my_ip*>.do**	Source directly with your simulator.
Synopsys VCS	**/simulation/modelsim/<*revision name*>_<*rtl or gate*>.vcs**	Add your testbench file name to this options file to pass the file to VCS using the `-file` option. If you specify a testbench file to NativeLink, NativeLink generates an **.sh** script that runs VCS.
Synopsys VCS MX	**/simulation/scsim/<*revision name*>_vcsmx_<*rtl or gate*>_<*verilog or vhdl*>.tcl**	Run this script at the command line using the command: `quartus_sh -t` <*script*> Any testbench you specify with NativeLink is included in this script.
Cadence Incisive (NC SIM)	**/simulation/ncsim/<*revision name*>_ncsim_<*rtl or gate*>_<*verilog or vhdl*>.tcl**	Run this script at the command line using the command: `quartus_sh -t` <*script*>. Any testbench you specify with NativeLink is included in this script.

You can use the following script variables:

- `TOP_LEVEL_NAME`—The top-level entity of your simulation is often a testbench that instantiates your design, and then your design instantiates IP cores and/or Qsys systems. Set the value of `TOP_LEVEL_NAME` to the top-level entity.
- `QSYS_SIMDIR`—Specifies the top-level directory containing the simulation files.
- Other variables control the compilation, elaboration, and simulation process.

Generating Custom Simulation Scripts with ip-make-simscript

Use the `ip-make-simscript` utility to generate simulation command scripts for multiple IP cores or Qsys systems. Specify all Simulation Package Descriptor files (**.spd**), each of which lists the required simulation files for the corresponding IP core or Qsys system. The IP parameter editor generates the **.spd** files.

`ip-make-simscript` compiles IP simulation models into various simulation libraries. Use the `compile-to-work` option to compile all simulation files into a single work library. Use this option only if you require a simplified library structure.

When you specify multiple **.spd** files, the `ip-make-simscript` utility generates a single simulation script containing all required simulation information. The default value of `TOP_LEVEL_NAME` is the `TOP_LEVEL_NAME` defined in the IP core or Qsys **.spd** file.

Set appropriate variables in the script, or edit the variable assignment directly in the script. If the simulation script is a Tcl file that is sourced in the simulator, set the variables before sourcing the script. If the simulation script is a shell script, pass in the variables as command-line arguments to the shell script.

- Type `ip-make-simscript` at the command prompt to run.
- Type `ip-make-simscript --help` for help on command options and syntax.

Table 1-9: ip-make-simscript Examples

Option	Description	Status
`--spd=`<*file*>	Describes the list of compiled files and memory model hierarchy. If your design includes multiple IP cores or Qsys systems that include **.spd** files, use this option for each file. You can specify multiple **.spd** files as a comma-separated list. For example: `ip-make-simscript --spd=,ip1.spd, ip2.spd,`	Required
`--output-directory=`<*directory*>	Specifies the location of output files. If unspecified, the default setting is the directory from which `ip-make-simscript` is run.	Optional
`--compile-to-work`	Compiles all design files to the default work library. Use this option only if you encounter problems managing your simulation with multiple libraries.	Optional
`--use-relative-paths`	Uses relative paths whenever possible.	Optional

Synthesizing Altera IP Cores in Other EDA Tools

You can use supported EDA tools to synthesize a design that includes Altera IP cores. When you generate the IP core synthesis files for use with third-party EDA synthesis tools, you can optionally create an area and timing estimation netlist. To enable generation, turn on **Create timing and resource estimates for third-party EDA synthesis tools** when customizing your IP variation.

The area and timing estimation netlist describes the IP core connectivity and architecture, but does not include details about the true functionality. This information enables certain third-party synthesis tools to better report area and timing estimates. In addition, synthesis tools can use the timing information to achieve timing-driven optimizations and improve the quality of results.

The Quartus II software generates the **<*variant name*>_syn.v** netlist file in Verilog HDL format regardless of the output file format you specify. If you use this netlist for synthesis, you must include the IP core wrapper file **<*variant name*>.v** or **<*variant name*>.vhd** in your Quartus II project.

Related Information

Quartus II Integrated Synthesis

Instantiating IP Cores in HDL

You can instantiate an IP core directly in your HDL code by calling the IP core name and declaring its parameters, in the same manner as any other module, component, or subdesign. When instantiating an IP core in VHDL, you must include the associated libraries.

Example Top-Level Verilog HDL Module

Verilog HDL ALTFP_MULT in Top-Level Module with One Input Connected to Multiplexer.

```
module MF_top (a, b, sel, datab, clock, result);
       input [31:0] a, b, datab;
       input clock, sel;
       output [31:0] result;
       wire [31:0] wire_dataa;

       assign wire_dataa = (sel)? a : b;
       altfp_mult inst1
(.dataa(wire_dataa), .datab(datab), .clock(clock), .result(result));

       defparam
               inst1.pipeline = 11,
               inst1.width_exp = 8,
               inst1.width_man = 23,
               inst1.exception_handling = "no";
endmodule
```

Example Top-Level VHDL Module

VHDL ALTFP_MULT in Top-Level Module with One Input Connected to Multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;
library altera_mf;
use altera_mf.altera_mf_components.all;

entity MF_top is
        port (clock, sel  : in  std_logic;
              a, b, datab : in  std_logic_vector(31 downto 0);
              result      : out std_logic_vector(31 downto 0));
end entity;

architecture arch_MF_top of MF_top is
signal wire_dataa : std_logic_vector(31 downto 0);
begin

wire_dataa <= a when (sel = '1') else b;

inst1 : altfp_mult
        generic map    (
                   pipeline => 11,
                   width_exp => 8,
                   width_man => 23,
                   exception_handling => "no")
        port map (
                   dataa => wire_dataa,
                   datab => datab,
                   clock => clock,
                   result => result);
end arch_MF_top;
```

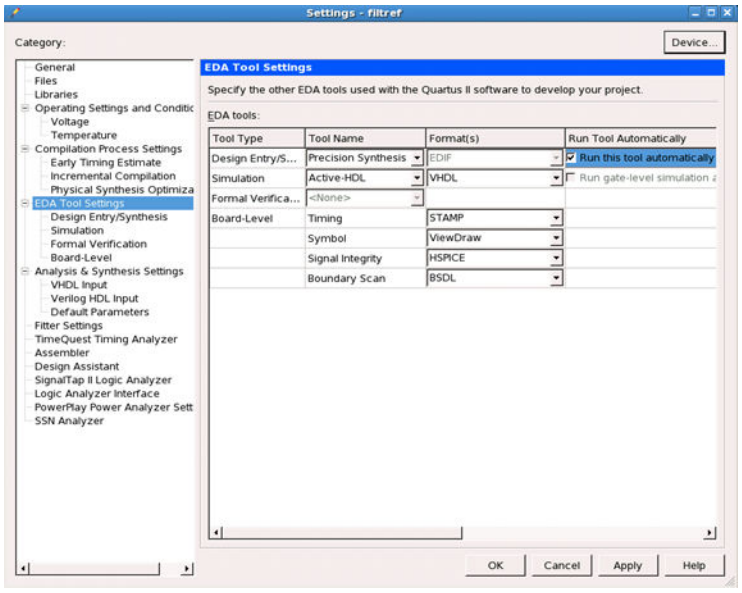
Integrating Other EDA Tools

You can integrate supported EDA design entry, synthesis, simulation, physical synthesis, and formal verification tools into the Quartus II design flow. The Quartus II software supports netlist files from other EDA design entry and synthesis tools. The Quartus II software optionally generates various files for use in other EDA tools.

The Quartus II software manages EDA tool files and provides the following integration capabilities:

- Automatically generate files for synthesis and simulation and automatically launch other EDA tools (**Assignments** > **Settings** > **EDA Tool Settings** > **NativeLink Settings**).
- Compile all RTL and gate-level simulation model libraries for your device, simulator, and design language automatically (**Tools** > **Launch Simulation Library Compiler**).
- Include files (**.edf**, **.vqm**) generated by other EDA design entry or synthesis tools in your project as synthesized design files (**Project** > **Add/Remove File from Project**) .
- Automatically generate optional filesfor board-level verification (**Assignments** > **Settings** > **EDA Tool Settings**).

Figure 1-22: EDA Tool Settings



EDA Tool Settings

Specify the other EDA tools used with the Quartus II software to develop your project.

EDA tools:

Tool Type	Tool Name	Format(s)	Run Tool Automatically
Design Entry/S...	Precision Synthesis	EDIF	[✓ Run this tool automatically]
Simulation	Active-HDL	VHDL	Run gate-level simulation a
Formal Verifica...	<None>		
Board-Level	Timing	STAMP	
	Symbol	ViewDraw	
	Signal Integrity	HSPICE	
	Boundary Scan	BSDL	

Related Information

Mentor Graphics Precision Synthesis SupportGraphics on page 18-1

Simulating Altera Designs

Managing Team-based Projects

The Quartus II software supports multiple designers, design iterations, and platforms. You can use the following techniques to preserve and track project changes in a team-based environment. These techniques may also be helpful for individual designers.

Related Information

- **Preserving Compilation Results** on page 1-36
- **Archiving Projects** on page 1-38
- **Using External Revision Control** on page 1-39
- **Migrating Projects Across Operating Systems** on page 1-40

Preserving Compilation Results

The Quartus II software maintains a database of compilation results for each project revision. The databases files store results of incremental or full compilation. Do not edit these files directly. However, you can use the database files in the following ways:

- Preserve compilation results for migration to a new version of the Quartus II software. Export a post-synthesis or post-fit, version-compatible database (**Project** > **Export Database**), and then import it into a newer version of the Quartus II software (**Project** > **Import Database**), or into another project.
- Optimize and lock down the compilation results for individual blocks. Export the post-synthesis or post-fit netlist as a Quartus II Exported Partition File (**.qxp**) (**Project** > **Export Design Partition**). You can then import the partition as a new project design file.
- Purge the content of the project database (**Project** > **Clean Project**) to remove unwanted previous compilation results at any time.

Factors Affecting Compilation Results

Changes to any of the following factors can impact compilation results:

- Project Files—project settings (**. qsf**), design files, and timing constraints (**.sdc**).
- Hardware—CPU architecture, not including hard disk or memory size differences. Windows XP x32 results are not identical to Windows XP x64 results. Linux x86 results is not identical to Linux x86_64.
- Quartus II Software Version—including build number and installed patches. Click **Help** > **About** to obtain this information.
- Operating System—Windows or Linux operating system, excluding version updates. For example, Windows XP, Windows Vista, and Windows 7 results are identical. Similarly, Linux RHEL, CentOS 4, and CentOS 5 results are identical.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1
- **Design Planning for Partial Reconfiguration** on page 4-1

The Partial Reconfiguration (PR) feature in the Quartus II software allows you to reconfigure a portion of the FPGA dynamically, while the remainder of the device continues to operate. The Quartus II software supports the PR feature for the Altera® Stratix® V device family.

Migrating Results Across Quartus II Software Versions

View basic information about your project in the Project Navigator, Report panel, and Messages window.

To preserve compilation results for migration to a later version of the Quartus II software, export a version-compatible database file, and then import it into the later version of the Quartus II software. A few device families do not support version-compatible database generation, as indicated by project messages.

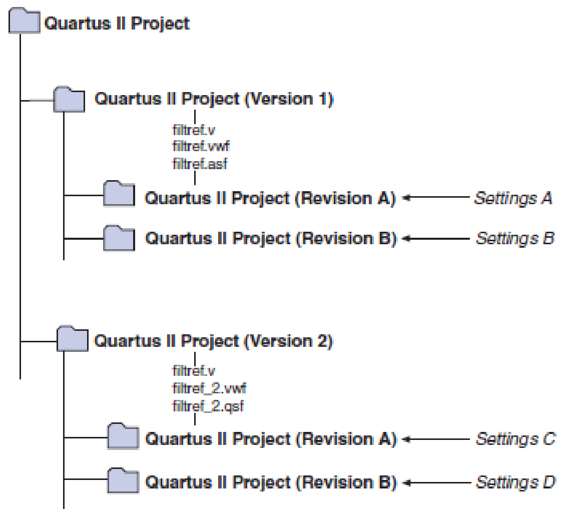
Exporting and Importing the Results Database

To save the compilation results in a version-compatible format for migration to a later version of the Quartus II software, follow these steps:

1. Open the project for migration in the original version of the Quartus II software.
2. Generate the project database and netlist with one of the following:
   - Click **Processing** > **Start** > **Start Analysis & Synthesis** to generate a post-synthesis netlist.
   - Click **Processing** > **Start Compilation** to generate a post-fit netlist.
3. Click **Project** > **Export Database** and specify the **Export directory**.
4. In a later version of the Quartus II software, click **New Project Wizard** and create a new project with the same top-level design entity name as the migrated project.
5. Click **Project** > **Import Database** and select the **<project directory>** **/export_db/exported database directory**. The Quartus II software opens the compiled project and displays compilation results.

Note: You can turn on **Assignments** > **Settings** > **Compilation Process Settings** > **Export version-compatible database** if you want to always export the database following compilation.

Figure 1-23: Quartus II Version-Compatible Database Structure


Quartus II Project
Quartus II Project (Version 1)
filtref.v
filtref.vwf
filtref.asf
Quartus II Project (Revision A)
Settings A
Quartus II Project (Revision B)
Settings B
Quartus II Project (Version 2)
filtref.v
filtref_2.vwf
filtref_2.qsf
Quartus II Project (Revision A)
Settings C
Quartus II Project (Revision B)
Settings D

Cleaning the Project Database

To clean the project database and remove all prior compilation results, follow these steps:

1. Click **Project** > **Clean Project**.
2. Select **All revisions** to remove the databases for all revisions of the current project, or specify a **Revision name** to remove only that revision's database.
3. Click **OK**. A message indicates when the database is clean.

Archiving Projects

You can save the elements of a project in a single, compressed Quartus II Archive File (**. qar**) by clicking **Project** > **Archive Project**.

The **.qar** captures logic design, project, and settings files required to restore the project.

Use this technique to share projects between designers, or to transfer your project to a new version of the Quartus II software, or to Altera support. You can optionally add compilation results, Qsys system files, and third-party EDA tool files to the archive. If you restore the archive in a different version of the Quartus II software, you must include the original **.qdf** in the archive to preserve original compilation results.

Manually Adding Files To Archives

To manually add files to an archive:

1. Click **Project** > **Archive Project** and specify the archive file name.
2. Click **Advanced**.
3. Select the **File set** for archive or select **Custom**. Turn on **File subsets** for archive.
4. Click **Add** and select Qsys system or EDA tool files. Click **OK**.
5. Click **Archive**.

Archiving Compilation Results

You can include compilation results in a project archive to avoid recompilation and preserve original results in the restored project. To archive compilation results, export the post-synthesis or post-fit version compatible database and include this file in the archive.

1. Export the project database.
2. Click **Project** > **Archive Project** and specify the archive file name.
3. Click **Advanced**.
4. Under **File subsets**, turn on **Version-compatible database files** and click **OK**.
5. Click **Archive**.

To restore an archive containing a version-compatible database, follow these steps:

1. Click **Project** > **Restore Archived Project**.
2. Select the archive name and destination folder and click **OK**.
3. After restoring the archived project, click **Project** > **Import Database** and import the version-compatible database.

Related Information

Exporting and Importing the Results Database on page 1-37

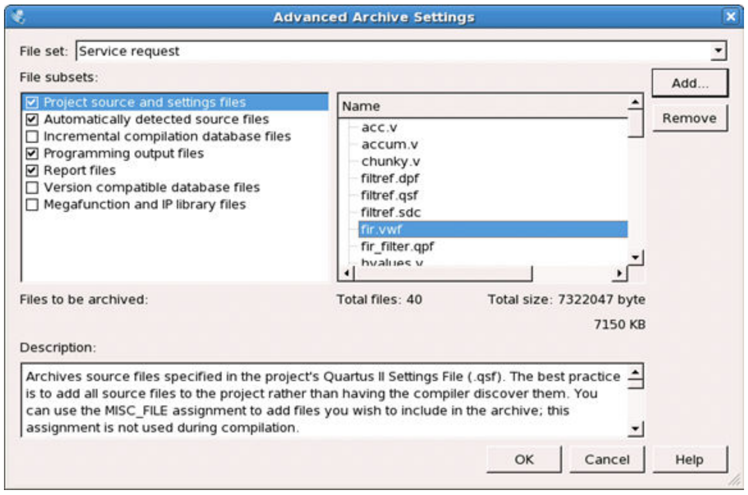
Archiving Projects for Altera Service Requests

When archiving projects for an Altera service request, include all of the following file types for proper debugging by Altera Support:

To quickly identify and include appropriate archive files for an Altera service request:

1. Click **Project** > **Archive Project** and specify the archive file name.
2. Click **Advanced**.
3. In **File set**, select **Service Request** to include files for Altera Support.
   - Project source and setting files (**.v, .vhd, .vqm, .qsf, .sdc, .qip, .qpf, .cmp, .sip**)
   - Automatically detected source files (various)
   - Programming output files (**.** jdi, .sof, .pof)
   - Report files (**.rpt, .pin, .summary, .smsg**)
   - Qsys system and IP files (**.qsys, . qip**)
4. Click **OK**, and then click **Archive**.

Figure 1-24: Archiving Project for Service Request


Advanced Archive Settings

File set: Service request

File subsets:

- ☑ Project source and settings files
- ☑ Automatically detected source files
- ☐ Incremental compilation database files
- ☑ Programming output files
- ☑ Report files
- ☐ Version compatible database files
- ☐ Megafunction and IP library files

Name
acc.v
accum.v
chunky.v
filtref.dpf
filtref.qsf
filtref.sdc
fir.vwf
fir_filter.qpf
hvalues.v

Files to be archived: Total files: 40 Total size: 7322047 byte
7150 KB

Description:
Archives source files specified in the project's Quartus II Settings File (.qsf). The best practice is to add all source files to the project rather than having the compiler discover them. You can use the MISC_FILE assignment to add files you wish to include in the archive; this assignment is not used during compilation.

Using External Revision Control

Your project may involve different team members with distributed responsibilities, such as sub-module design, device and system integration, simulation, and timing closure. In such cases, it may be useful to track and protect file revisions in an external revision control system.

While Quartus II project revisions preserve various project setting and constraint combinations, external revision control systems can also track and merge RTL source code, simulation testbenches, and build scripts. External revision control supports design file version experimentation through branching and merging different versions of source code from multiple designers. Refer to your external revision control documentation for setup information.

Files to Include In External Revision Control

Include the following Quartus project file types in external revision control systems:

- Logic design files (**.v, .vdh, .bdf, edf, .vqm**)
- Timing constraint files (.sdc)
- Quartus project settings and constraints (**.qdf, .qpf, .qsf**)
- IP files (**.v, .sv, .vhd, .qip, .sip, .qsys**)
- Qsys-generated files (**.qsys, .qip, .sip**)
- EDA tool files (**.vo, .vho**)

You can generate or modify these files manually if you use a scripted design flow. If you use an external source code control system, you can check-in project files anytime you modify assignments and settings in the Quartus software.

Migrating Projects Across Operating Systems

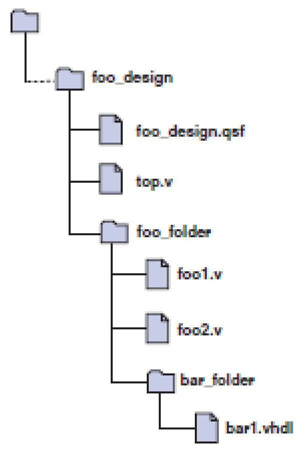
Consider the following cross-platform issues when moving your project from one operating system to another (for example, from Windows to Linux).

Migrating Design Files and Libraries

Consider the following file naming differences when migrating projects across operating systems:

- Use appropriate case for your platform in file path references.
- Use a character set common to both platforms.
- Do not change the forward-slash (`/`) and back-slash (`\`) path separators in the **.qsf**. The Quartus II software automatically changes all back-slash (`\`) path separators to forward-slashes (`/`)in the **.qsf**.
- Observe the target platform's file name length limit.
- Use underscore instead of spaces in file and directory names.
- Change library absolute path references to relative paths in the **.qsf**.
- Ensure that any external project library exists in the new platform's file system.
- Specify file and directory paths as relative to the project directory. For example, for a project titled **foo_design** , specify the source files as: **top.v**, **foo_folder /foo1.v**, **foo_folder /foo2.v**, and **foo_folder/ bar_folder/bar1.vhdl**.
- Ensure that all the subdirectories are in the same hierarchical structure and relative path as in the original platform.

Figure 1-25: All Inclusive Project Directory Structure

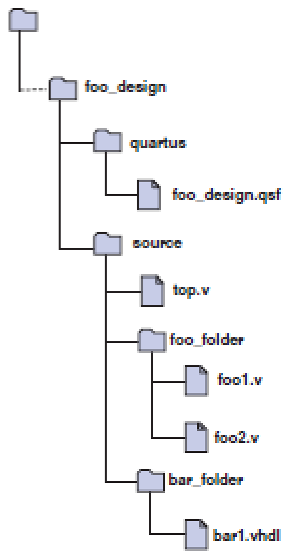


Use Relative Paths

Express file paths using relative path notation (**..** /).

For example, in the directory structure shown you can specify **top.v** as **../source/top.v** and **foo1.v** as **../ source/foo_folder/foo1.v**.

Figure 1-26: Quartus II Project Directory Separate from Design Files



Design Library Migration Guidelines

The following guidelines apply to library migration across computing platforms:

1. The project directory takes precedence over the project libraries.
2. For Linux, the Quartus II software creates the file in the **altera.quartus** directory under the <*home*> directory.
3. All library files are relative to the libraries. For example, if you specify the **user_lib1** directory as a project library and you want to add the **/user_lib1/foo1.v** file to the library, you can specify the **foo1.v** file in the **.qsf** as **foo1.v**. The Quartus II software includes files in specified libraries.
4. If the directory is outside of the project directory, an absolute path is created by default. Change the absolute path to a relative path before migration.
5. When copying projects that include libraries, you must either copy your project library files along with the project directory or ensure that your project library files exist in the target platform.
    - On Windows, the Quartus II software searches for the **quartus2.ini** file in the following directories and order:
    - USERPROFILE, for example, **C:\Documents and Settings** <*user name*>
    - Directory specified by the TMP environmental variable
    - Directory specified by the TEMP environmental variable
    - Root directory, for example, C:\

Scripting API

You can use command-line executables or scripts to execute project commands, rather than using the GUI. The following commands are available for scripting project management.

Scripting Project Settings

You can use a Tcl script to specify settings and constraints, rather than using the GUI. This can be helpful if you have many settings and wish to track them in a single file or spreadsheet for iterative comparison.

The **.qsf** supports only a limited subset of Tcl commands. Therefore, pass settings and constraints using a Tcl script:

1. Create a text file with the extension**.tcl** that contains your assignments in Tcl format.
2. Source the Tcl script file by adding the following line to the .qsf: `set_global_assignment -name SOURCE_TCL_SCR IPT_FILE <file name>`.

Project Revision Commands

Use the following commands for scripting project revisions.

Create Revision Command on page 1-42

Set Current Revision Command on page 1-42

Get Project Revisions Command on page 1-42

Delete Revision Command on page 1-42

Create Revision Command

```
create_revision <name> -based_on <revision_name> -copy_results -set_current
```

Option	Description
`based_on` (optional)	Specifies the revision name on which the new revision bases its settings.
`copy_results`	Copies the results from the "based_on" revision.
`set_current` (optional)	Sets the new revision as the current revision.

Set Current Revision Command

The `-force` option enables you to open the revision that you specify under revision name and overwrite the compilation database if the database version is incompatible.

```
set_current_revision -force <revision name>
```

Get Project Revisions Command

```
get_project_revisions <project_name>
```

Delete Revision Command

```
delete_revision <revision name>
```

Project Archive Commands

You can use Tcl commands and the `quartus_sh` executable to create and manage archives of a Quartus project.

Creating a Project Archive

in a Tcl script or from a Tcl prompt, you can use the following command to create a Quartus archive:

```
project_archive <name>.qar
```

You can specify the following other options:

- `-all_revisions` - Includes all revisions of the current project in the archive.
- `-auto_common_directory` - Preserves original project directory structure in archive
- `-common_directory /<name>` - Preserves original project directory structure in specified subdirectory
- `-include_libraries` - Includes libraries in archive
- `-include_outputs` - Includes output files in archive
- `-use_file_set <file_set>` Includes specified fileset in archive
- `-version_compatible_database` - Includes version-compatible database files in archive

Note: Version-compatible databases are not available for some device families. If you require the database files to reproduce the compilation results in the same Quartus software version, use the `-use_file_set full_db` option to archive the complete database.

Restoring an Archived Project

Use the following Tcl command to restore a Quartus project:

```
project_restore <name>.qar -destination restored -overwrite
```

This example restores to a destination directory named "restored".

Project Database Commands

Use the following commands for managing Quartus project databases:

Import and Export Version-Compatible Databases on page 1-43

Import and Export Version-Compatible Databases from a Flow Package on page 1-43

Generate Version-Compatible Database After Compilation on page 1-44

quartus_cdb and quartus_sh Executables to Manage Version-Compatible Databases on page 1-44

Import and Export Version-Compatible Databases

Use the following commands to import or export a version-compatible database:

- `import_database <directory>`
- `export_database <directory>`

Import and Export Version-Compatible Databases from a Flow Package

The following are Tcl commands from the `flow` package to import or export version-compatible databases. If you use the `flow` package, you must specify the database directory variable name. `flow` and `database_manager` packages contain commands to manage version-compatible databases.

- `set_global_assignment -name VER_COMPATIBLE_DB_DIR <directory>`
- *`execute_flow –flow export_database`*
- execute_flow –flow import_database

Generate Version-Compatible Database After Compilation

Use the following commands to generate a version-compatible database after compilation:

- `set_global_assignment -name AUTO_EXPORT_VER_COMPATIBLE_DB ON`
- `set_global_assignment-name VER_COMPATIBLE_DB_DIR <directory>`

quartus_cdb and quartus_sh Executables to Manage Version-Compatible Databases

Use the following commands to manage version-compatible databases:

- `quartus_cdb <project> -c <revision>--export_database=<directory>`
- `quartus_cdb <project> -c <revision> --import_database=<directory>`
- `quartus_sh –flow export_database <project> -c \ <revision>`
- `quartus_sh –flow import_database <project> -c \ <revision>`

Project Library Commands

Use the following commands to script project library changes.

Specify Project Libraries With SEARCH_PATH Assignment on page 1-44

Report Specified Project Libraries Commands on page 1-44

Related Information

- **Tcl Scripting**
- **CommandLine Scripting**
- **Quartus Settings File Manual**

Specify Project Libraries With SEARCH_PATH Assignment

In Tcl, use commands in the `:: quartus ::project` package to specify project libraries, and the `set_global_assignment` command.

Use the following commands to script project library changes:

- `set_global_assignment -name SEARCH_PATH "../other_dir/library1"`
- `set_global_assignment -name SEARCH_PATH "../other_dir/library2"`
- `set_global_assignment -name SEARCH_PATH "../other_dir/library3"`

Report Specified Project Libraries Commands

To report any project libraries specified for a project and any global libraries specified for the current installation of the Quartus software, use the `get_global_assignment` and `get_user_option` Tcl commands.

Use the following commands to report specified project libraries:

- `get_global_assignment -name SEARCH_PATH`
- `get_user_option -name SEARCH_PATH`

Document Revision History

Table 1-10: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• Added description of design templates feature. • Updated screenshot for DSE II GUI. • Added qsys_script IP core instantiation information. • Described changes to generating and processing of instance and entity names. • Added description of upgrading IP cores at the command line. • Updated procedures for upgrading and migrating IP cores. • Gate level timing simulation supported only for Cyclone IV and Stratix IV devices.
2014.12.15	14.1.0	• Updated content for DSE II GUI and optimizations. • Added information about new **Assignments** > **Settings** > **IP Settings** that control frequency of synthesis file regeneration and automatic addtion of IP files to the project.
2014.08.18	14.0a10.0	• Added information about specifying parameters for IP cores targeting Arria 10 devices. • Added information about the latest IP output for Quartus II version 14.0a10 targeting Arria 10 devices. • Added information about individual migration of IP cores to the latest devices. • Added information about editing existing IP variations.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog. • Added standard information about upgrading IP cores. • Added standard installation and licensing information. • Removed outdated device support level information. IP core device support is now available in IP Catalog and parameter editor.
November 2013	13.1.0	• Conversion to DITA format
May 2013	13.0.0	• Overhaul for improved usability and updated information.
June 2012	12.0.0	• Removed survey link. • Updated information about `VERILOG_INCLUDE_FILE`.
November 2011	10.1.1	Template update.

Date	Version	Changes
December 2010	10.1.0	• Changed to new document template. • Removed Figure 4–1, Figure 4–6, Table 4–2. • Moved "Hiding Messages" to Help. • Removed references about the `set_user_option` command. • Removed Classic Timing Analyzer references.

Related Information

Quartus II Handbook Archive

Design Planning with the Quartus II Software 2

2015.05.04

Design Planning with the Quartus II Software

Because of the significant increase in FPGA device densities, designs are complex and can sometimes involve multiple designers. System architects must also resolve design issues when integrating design blocks. However, you can solve potential problems early in the design cycle by following the design planning considerations in this chapter.This chapter provides only an introduction to various design planning features in the Quartus® II software.

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

The Quartus II software optimizes designs for the best overall results; however, you can change the settings to better optimize one aspect of your design, such as power utilization. Certain tools or debugging options can lead to restrictions in your design flow. Your design priorities help you choose the tools, features, and methodologies to use for your design.

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

Creating Design Specifications

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

In addition, creating a test plan helps you to design for verification and ease of manufacture. For example, you might need to validate interfaces incorporated in your design. To perform any built-in self-test functions to drive interfaces, you can use a UART interface with a Nios® II processor inside the FPGA device.

If more than one designer works on your design, you must consider a common design directory structure or source control system to make design integration easier. Consider whether you want to standardize on an interface protocol for each design block.

Related Information

- **Planning for On-Chip Debugging Tools** on page 2-8
  For guidelines related to analyzing and debugging the device after it is in the system.
- **Planning for Hierarchical and Team-Based Design** on page 2-11
  For more suggestions on team-based designs.
- **Using Qsys and Standard Interfaces in System Design** on page 2-2
  For improved reusability and ease of integration.

Selecting Intellectual Property

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

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

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

Related Information

Intellectual Property
For descriptions of available IP cores.

Using Qsys and Standard Interfaces in System Design

You can use the Quartus II Qsys system integration tool to create your design with fast and easy system-level integration. With Qsys, you can specify system components in a GUI and generate the required interconnect logic automatically, along with adapters for clock crossing and width differences.

Because system design tools change the design entry methodology, you must plan to start developing your design within the tool. Ensure all design blocks use appropriate standard interfaces from the beginning of the design cycle so that you do not need to make changes later.

Qsys components use Avalon® standard interfaces for the physical connection of components, and you can connect any logical device (either on-chip or off-chip) that has an Avalon interface. The Avalon Memory-Mapped interface allows a component to use an address mapped read or write protocol that enables flexible topologies for connecting master components to any slave components. The Avalon Streaming interface enables point-to-point connections between streaming components that send and receive data using a high-speed, unidirectional system interconnect between source and sink ports.

In addition to enabling the use of a system integration tool such as Qsys, using standard interfaces ensures compatibility between design blocks from different design teams or vendors. Standard interfaces simplify the interface logic to each design block and enable individual team members to test their individual design blocks against the specification for the interface protocol to ease system integration.

Related Information

- System Design with Qsys
  For more information about using Qsys to improve your productivity.
- SOPC Builder User Guide
  For more information about SOPC Builder.

Device Selection

The device you choose affects board specification and layout. This section provides guidelines in the device selection process.

Choose the device family that best suits your design requirements. Families differ in cost, performance, logic and memory density, I/O density, power utilization, and packaging. You must also consider feature requirements, such as I/O standards support, high-speed transceivers, global or regional clock networks, and the number of phase-locked loops (PLLs) available in the device.

Each device family also has a device handbook, including a data sheet, which documents device features in detail. You can also see a summary of the resources for each device in the **Device** dialog box in the Quartus II software.

Carefully study the device density requirements for your design. Devices with more logic resources and higher I/O counts can implement larger and more complex designs, but at a higher cost. Smaller devices use lower static power. Select a device larger than what your design requires if you want to add more logic later in the design cycle to upgrade or expand your design, and reserve logic and memory for on-chip debugging. Consider requirements for types of dedicated logic blocks, such as memory blocks of different sizes, or digital signal processing (DSP) blocks to implement certain arithmetic functions.

If you have older designs that target an Altera device, you can use their resources as an estimate for your design. Compile existing designs in the Quartus II software with the **Auto device selected by the Fitter** option in the **Settings** dialog box. Review the resource utilization to learn which device density fits your design. Consider coding style, device architecture, and the optimization options used in the Quartus II software, which can significantly affect the resource utilization and timing performance of your design.

Related Information

- Planning for On-Chip Debugging Tools on page 2-8
  For information about on-chip debugging.
- Altera Product Selector
  For You can refer to the Altera website to help you choose your device.
- Selector Guides
  You can review important features of each device family in the refer to the Altera website.
- Devices and Adapters
  For a list of device selection guides.
- IP and Megafunctions
  For information on how to obtain resource utilization estimates for certain configurations of Altera's IP, refer to the user guides for Altera megafunctions and IP MegaCores on the literature page of the Altera website.

Device Migration Planning

Determine whether you want to migrate your design to another device density to allow flexibility when your design nears completion. You may want to target a smaller (and less expensive) device and then move to a larger device if necessary to meet your design requirements. Other designers may prototype their design in a larger device to reduce optimization time and achieve timing closure more quickly, and then migrate to a smaller device after prototyping. If you want the flexibility to migrate your design, you must specify these migration options in the Quartus II software at the beginning of your design cycle.

Selecting a migration device impacts pin placement because some pins may serve different functions in different device densities or package sizes. If you make pin assignments in the Quartus II software, the Pin Migration View in the Pin Planner highlights pins that change function between your migration devices.

Related Information

- **Early Pin Planning and I/O Analysis** on page 2-5
- **Specifying Devices for Device Migration**
  For more information about specifying the target migration devices in Quartus II Help.

Planning for Device Programming or Configuration

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

The device family handbooks describe the configuration options available for a device family. For information about programming CPLD devices, refer to your device data sheet or handbook.

Related Information

Configuration Handbook
For more details about configuration options.

Estimating Power

You can use the Quartus II power estimation and analysis tools to provide information to PCB board and system designers. Power consumption in FPGA devices depends on the design logic, which can make planning difficult. You can estimate power before you create any source code, or when you have a preliminary version of the design source code, and then perform the most accurate analysis with the PowerPlay Power Analyzer when you complete your design.

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

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

The PowerPlay Early Power Estimator (EPE) spreadsheet allows you to estimate power utilization for your design.

You can manually enter data into the EPE spreadsheet, or use the Quartus II software to generate device resource information for your design.

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

If you have an existing design or a partially completed design, you can use the Quartus II software to generate the PowerPlay Early Power Estimator File (**.txt**, **.csv**) to assist you in completing the PowerPlay EPE spreadsheet.

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

Estimating power consumption early in the design cycle allows planning of power budgets and avoids unexpected results when designing the PCB.

When you complete your design, perform a complete power analysis to check the power consumption more accurately. The PowerPlay Power Analyzer tool in the Quartus II software provides an accurate estimation of power, ensuring that thermal and supply limitations are met.

Related Information

- **PowerPlay Power Analysis**
  For more information about power estimation and analysis.
- **Performing an Early Power Estimate Using the PowerPlay Early Power Estimator**
  For more information about generating the PowerPlay EPE File, refer to Quartus II Help.
- **PowerPlay Early Power Estimator and Power Analyzer**
  The PowerPlay EPE spreadsheets for each supported device family are available on the Altera website.

Early Pin Planning and I/O Analysis

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

You can create a preliminary pin-out for an Altera FPGA with the Quartus II Pin Planner before you develop the source code, based on standard I/O interfaces (such as memory and bus interfaces) and any other I/O requirements for your system. The Quartus II I/O Assignment Analysis checks that the pin locations and assignments are supported in the target FPGA architecture. You can then use I/O Assignment Analysis to validate I/O-related assignments that you create or modify throughout the design process. When you compile your design in the Quartus II software, I/O Assignment Analysis runs automatically in the Fitter to validate that the assignments meet all the device requirements and generates error messages.

Early in the design process, before creating the source code, the system architect has information about the standard I/O interfaces (such as memory and bus interfaces), the IP cores in your design, and any other I/O-related assignments defined by system requirements. You can use this information with the **Early Pin Planning** feature in the Pin Planner to specify details about the design I/O interfaces. You can then create a top-level design file that includes all I/O information.

The Pin Planner interfaces with the IP core parameter editor, which allows you to create or import custom IP cores that use I/O interfaces. You can configure how to connect the functions and cores to each other by specifying matching node names for selected ports. You can create other I/O-related assignments for these interfaces or other design I/O pins in the Pin Planner, as described in this section. The Pin Planner creates virtual pin assignments for internal nodes, so internal nodes are not assigned to device pins during compilation. After analysis and synthesis of the newly generated top-level wrapper file, use the generated netlist to perform I/O Analysis with the **Start I/O Assignment Analysis** command.

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

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

With the Pin Planner, you can identify I/O banks, voltage reference (VREF) groups, and differential pin pairings to help you through the I/O planning process. If you selected a migration device, the **Pin Migration View** highlights the pins that have changed functions in the migration device when compared to the currently selected device. Selecting the pins in the Device Migration view cross-probes to the rest of the Pin Planner, so that you can use device migration information when planning your pin assignments. You can also configure board trace models of selected pins for use in "board-aware" signal integrity reports generated with the **Enable Advanced I/O Timing** option . This option ensures that you get accurate I/O timing analysis. You can use a Microsoft Excel spreadsheet to start the I/O planning process if you normally use a spreadsheet in your design flow, and you can export a Comma-Separated Value File (**.csv**) containing your I/O assignments for spreadsheet use when you assign all pins.

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

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

Related Information

- Device Migration Planning on page 2-3

- **Set Up Top-Level Design File Window (Edit Menu**
  For more information about setting up the nodes in your design, refer to Quartus II Help.
- **I/O Management**
  For more information about I/O assignment and analysis.
- **Mentor Graphics PCB Design Tools Support**
- **Cadence PCB Design Tools Support**
  For more information about passing I/O information between the Quartus II software and third-party EDA tools.

Simultaneous Switching Noise Analysis

Simultaneous switching noise (SSN) is a noise voltage inducted onto a victim I/O pin of a device due to the switching behavior of other aggressor I/O pins in the device.

Altera provides tools for SSN analysis and estimation, including SSN characterization reports, an Early SSN Estimator (ESE) spreadsheet tool, and the SSN Analyzer in the Quartus II software. SSN often leads to the degradation of signal integrity by causing signal distortion, thereby reducing the noise margin of a system. You must address SSN with estimation early in your system design, to minimize later board design changes. When your design is complete, verify your board design by performing a complete SSN analysis of your FPGA in the Quartus II software.

Related Information

- **Altera's Signal Integrity Center**
  For more information and device support for the ESE spreadsheet tool on the Altera website.
- **Simultaneous Switching Noise (SSN**
  For more information about the SSN Analyzer.

Selecting Third-Party EDA Tools

Your complete FPGA design flow may include third-party EDA tools in addition to the Quartus II software. Determine which tools you want to use with the Quartus II software to ensure that they are supported and set up properly, and that you are aware of their capabilities.

Synthesis Tool

The Quartus II software includes integrated synthesis that supports Verilog HDL, VHDL, Altera Hardware Description Language (AHDL), and schematic design entry.

You can also use supported standard third-party EDA synthesis tools to synthesize your Verilog HDL or VHDL design, and then compile the resulting output netlist file in the Quartus II software. Different synthesis tools may give different results for each design. To determine the best tool for your application, you can experiment by synthesizing typical designs for your application and coding style. Perform placement and routing in the Quartus II software to get accurate timing analysis and logic utilization results.

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

Tool vendors frequently add new features, fix tool issues, and enhance performance for Altera devices, you must use the most recent version of third-party synthesis tools.

Related Information

- **Incremental Compilation with Design Partitions** on page 2-12
  For more information on how to use incremental compilation with design partitions.
- **Volume 1: Design and Synthesis**
  For more information about synthesis tool flows.
- **Quartus II Software and Device Support Release Notes**
  For more information about the supported versions of synthesis tools in your version of the Quartus II Help.

Simulation Tool

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

Use the simulator version that your Quartus II software version supports for best results. You must also use the model libraries provided with your Quartus II software version. Libraries can change between versions, which might cause a mismatch with your simulation netlist.

Related Information

- **Quartus II Software and Device Support Release Notes**
  For a list of the version of each simulation tool that is supported with a given version of the Quartus II software.
- **Simulation**
  For more information about simulation tool flows.

Formal Verification Tools

Consider whether the Quartus II software supports the formal verification tool that you want to use, and whether the flow impacts your design and compilation stages of your design.

Using a formal verification tool can impact performance results because performing formal verification requires turning off certain logic optimizations, such as register retiming, and forces you to preserve hierarchy blocks, which can restrict optimization. Formal verification treats memory blocks as black boxes. Therefore, you must keep memory in a separate hierarchy block so other logic does not get incorporated into the black box for verification. If formal verification is important to your design, plan for limitations and restrictions at the beginning of the design cycle rather than make changes later.

Related Information

Volume 3: Verification
For more information about formal verification flows and the supported tools

Planning for On-Chip Debugging Tools

In-system debugging tools offer different advantages and trade-offs. A particular debugging tool may work better for different systems and designers.

You must evaluate on-chip debugging tools early in your design process, to ensure that your system board, Quartus II project, and design can support the appropriate tools. You can reduce debugging time and avoid making changes to accommodate your preferred debugging tools later.

If you intend to use any of these tools, you may have to plan for the tools when developing your system board, Quartus II project, and design. Consider the following debugging requirements when you plan your design:

- JTAG connections—required to perform in-system debugging with JTAG tools. Plan your system and board with JTAG ports that are available for debugging.
- Additional logic resources—required to implement JTAG hub logic. If you set up the appropriate tool early in your design cycle, you can include these device resources in your early resource estimations to ensure that you do not overload the device with logic.
- Reserve device memory—required if your tool uses device memory to capture data during system operation. To ensure that you have enough memory resources to take advantage of this debugging technique, consider reserving device memory to use during debugging.
- Reserve I/O pins—required if you use the Logic Analyzer Interface (LAI) or SignalProbe tools, which require I/O pins for debugging. If you reserve I/O pins for debugging, you do not have to later change your design or board. The LAI can multiplex signals with design I/O pins if required. Ensure that your board supports a debugging mode, in which debugging signals do not affect system operation.
- Instantiate an IP core in your HDL code—required if your debugging tool uses an Altera IP core.
- Instantiate the SignalTap II Logic Analyzer IP core—required if you want to manually connect the SignalTap II Logic Analyzer to nodes in your design and ensure that the tapped node names do not change during synthesis. You can add the analyzer as a separate design partition for incremental compilation to minimize recompilation times.

Table 2-1: Factors to Consider When Using Debugging Tools During Design Planning Stages

Design Planning Factor	SignapTap II Logic Analyzer	System Console	In-System Memory Content Editor	Logic Analyzer Interface (LAI)	SignalP-robe	In-System Sources and Probes	Virtual JTAG IP Core
JTAG connections	Yes	Yes	Yes	Yes	—	Yes	Yes
Additional logic resources	—	Yes	—	—	—	—	Yes
Reserve device memory	Yes	Yes	—	—	—	—	—
Reserve I/O pins	—	—	—	Yes	Yes	—	—
Instantiate IP core in your HDL code	—	—	—	—	—	Yes	Yes

Related Information

- **System Debugging Tools Overview**
  For an overview of debugging tools that can help you decide which tools to use.
- **Design Debugging Using the SignalTap II Logic Analyzer**
  For more information on using the SignalTap II Logic Analyzer.

Design Practices and HDL Coding Styles

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

Design Recommendations

Use synchronous design practices to consistently meet your design goals. Problems with asynchronous design techniques include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. When you meet all register timing requirements, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades.

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

The Design Assistant in the Quartus II software is a design-rule checking tool that enables you to verify design issues. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. You can also use third-party lint tools to check your coding style.

Consider the architecture of the device you choose so that you can use specific features in your design. For example, the control signals should use the dedicated control signals in the device architecture. Sometimes, you might need to limit the number of different control signals used in your design to achieve the best results.

Related Information

- **About the Design Assistant**
  For more information about running the Design Assistant, refer to Quartus II Help.
- **Recommended Design Practices** on page 11-1
  For more information about design recommendations and using the Design Assistant.
- **www.sunburst-design.com**
  You can also refer to industry papers for more information about multiple clock design. For a good analysis, refer to *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs* under **Papers**.

Recommended HDL Coding Styles

HDL coding styles can have a significant effect on the quality of results for programmable logic designs.

If you design memory and DSP functions, you must understand the target architecture of your device so you can use the dedicated logic block sizes and configurations. Follow the coding guidelines for inferring megafunctions and targeting dedicated device hardware, such as memory and DSP blocks.

Related Information

- **Recommended HDL Coding Styles** on page 12-1
  For HDL coding examples and recommendations, refer to the Recommended HDL Coding Styles chapter in volume 1 of the Quartus II Handbook. For additional tool-specific guidelines

Managing Metastability

Metastability problems can occur in digital design when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the designer cannot guarantee that the signal meets the setup and hold time requirements during the signal transfer.

Designers commonly use a synchronization chain to minimize the occurrence of metastable events. Ensure that your design accounts for synchronization between any asynchronous clock domains. Consider using a synchronizer chain of more than two registers for high-frequency clocks and frequently-toggling data signals to reduce the chance of a metastability failure.

You can use the Quartus II software to analyze the average mean time between failures (MTBF) due to metastability when a design synchronizes asynchronous signals, and optimize your design to improve the metastability MTBF. The MTBF due to metastability is an estimate of the average time between instances when metastability could cause a design failure. A high MTBF (such as hundreds or thousands of years between metastability failures) indicates a more robust design. Determine an acceptable target MTBF given the context of your entire system and the fact that MTBF calculations are statistical estimates.

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

Related Information

- **Managing Metastability with the Quartus II Software** on page 13-1
  For information about metastability analysis, reporting, and optimization features in the Quartus II software

Planning for Hierarchical and Team-Based Design

To create a hierarchical design so that you can use compilation-time savings and performance preservation with the Quartus II software incremental compilation feature, plan for an incremental compilation flow from the beginning of your design cycle. The following subsections describe the flat compilation flow, in which the design hierarchy is flattened without design partitions, and then the incremental compilation flow that uses design partitions. Incremental compilation flows offer several advantages, but require more design planning to ensure effective results. The last subsections discuss planning an incremental compilation flow, planning design partitions, and optionally creating a design floorplan.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1
  For information about using the incremental compilation flow methodology in the Quartus II software.

Flat Compilation Flow with No Design Partitions

In the flat compilation flow with no design partitions in the Quartus II software, the Quartus II software compiles the entire design in a “flat” netlist.

Your source code can have hierarchy, but the Quartus II software flattens your design during compilation and synthesizes all the design source code and fits in the target device whenever the software recompile your design after any change in your design. By processing the entire design, the software performs all available logic and placement optimizations on the entire design to improve area and performance. You can use debugging tools in an incremental design flow, such as the SignalTap II Logic Analyzer, but you do not specify any design partitions to preserve design hierarchy during compilation.

The flat compilation flow is easy to use; you do not have to plan any design partitions. However, because the Quartus II software recompiles the entire design whenever you change your design, compilation times can be slow for large devices. Additionally, you may find that the results for one part of the design change when you change a different part of your design. You run **Rapid Recompile** to preserve portions of previous placement and routing in subsequent compilations. This option can reduce your compilation time in a flat or partitioned design when you make small changes to your design.

Incremental Compilation with Design Partitions

In an incremental compilation flow, the system architect splits a large design into partitions. When hierarchical design partitions are well chosen and placed in the device floorplan, you can speed up your design compilation time while maintaining the quality of results.

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

Third-party IP designers can also export logic blocks to be integrated into the top-level design. Team members can work on partitions independently, which can simplify the design process and reduce compilation time. With exported partitions, the system architect must provide guidance to designers or IP providers to ensure that each partition uses the appropriate device resources. Because the designs may be developed independently, each designer has no information about the overall design or how their partition connects with other partitions. This lack of information can lead to problems during system integration. The top-level project information, including pin locations, physical constraints, and timing requirements, must be communicated to the designers of lower-level partitions before they start their design.

The system architect plans design partitions at the top level and allows third-party designs to access the top-level project framework. By designing in a copy of the top-level project (or by checking out the project files in a source control environment), the designers of the lower-level block have full information about the entire project, which helps to ensure optimal results.

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

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1
  For more information about support for Quartus II incremental compilation.

Planning Design Partitions and Floorplan Location Assignments

Partitioning a design for an FPGA requires planning to ensure optimal results when you integrate the partitions. Following Altera's recommendations for creating design partitions should improve the overall quality of results. For example, registering partition I/O boundaries keeps critical timing paths inside one partition that can be optimized independently. When you specify the design partitions, you can use the Incremental Compilation Advisor to ensure that partitions meet Altera's recommendations.

If you have timing-critical partitions that are changing through the design flow, or partitions exported from another Quartus II project, you can create design floorplan assignments to constrain the placement of the affected partitions. Good partition and floorplan design helps partitions meet top-level design requirements when integrated with the rest of your design, reducing time you spend integrating and verifying the timing of the top-level design.

Related Information

Best Practices for Incremental Compilation Partitions and Floorplan Assignments on page 14-1
For detailed guidelines about creating design partitions and organizing your source code, as well as information about when and how to create floorplan assignments .

Analyzing and Optimizing the Design Floorplan
For more information about creating floorplan assignments in the Chip Planner .

Running Fast Synthesis

You save time when you find design issues early in the design cycle rather than in the final timing closure stages. When the first version of the design source code is complete, you might want to perform a quick compilation to create a kind of silicon virtual prototype (SVP) that you can use to perform timing analysis.

If you synthesize with the Quartus II software, you can choose to perform a **Fast** synthesis, which reduces the compilation time, but may give reduced quality of results.

If you design individual design blocks or partitions separately, you can use the Fast synthesis and early timing estimate features as you develop your design. Any issues highlighted in the lower-level design blocks are communicated to the system architect. Resolving these issues might require allocating additional device resources to the individual partition, or changing the timing budget of the partition.

Expert designers can also use fast synthesis to prototype the entire design. Incomplete partitions are marked as empty in an incremental compilation flow, while the rest of the design is compiled to detect any problems with design integration.

Related Information

Synthesis Effort logic option
For more information about Fast synthesis, refer to Quartus II Help.

Document Revision History

Table 2-2: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	Remove support for Early Timing Estimate feature.
2014.06.30	14.0.0	Updated document format.
November 2013	13.1.0	Removed HardCopy device information.

Date	Version	Changes
November, 2012	12.1.0	Update for changes to early pin planning feature
June 2012	12.0.0	Editorial update.
November 2011	11.0.1	Template update.
May 2011	11.0.0	• Added link to System Design with Qsys in "Creating Design Specifications" on page 1–2 • Updated "Simultaneous Switching Noise Analysis" on page 1–8 • Updated "Planning for On-Chip Debugging Tools" on page 1–10 • Removed information from "Planning Design Partitions and Floorplan Location Assignments" on page 1–15
December 2010	10.1.0	• Changed to new document template • Updated "System Design and Standard Interfaces" on page 1–3 to include information about the Qsys system integration tool • Added link to the Altera Product Selector in "Device Selection" on page 1–3 • Converted information into new table (Table 1–1) in "Planning for On-Chip Debugging Options" on page 1–10 • Simplified description of incremental compilation usages in "Incremental Compilation with Design Partitions" on page 1–14 • Added information about the Rapid Recompile option in "Flat Compilation Flow with No Design Partitions" on page 1–14 • Removed details and linked to Quartus II Help in "Fast Synthesis and Early Timing Estimation" on page 1–16
July 2010	10.0.0	• Added new section "System Design" on page 1–3 • Removed details about debugging tools from "Planning for On-Chip Debugging Options" on page 1–10 and referred to other handbook chapters for more information • Updated information on recommended design flows in "Incremental Compilation with Design Partitions" on page 1–14 and removed "Single-Project Versus Multiple-Project Incremental Flows" heading • Merged the "Planning Design Partitions" section with the "Creating a Design Floorplan" section. Changed heading title to "Planning Design Partitions and Floorplan Location Assignments" on page 1–15 • Removed "Creating a Design Floorplan" section • Removed "Referenced Documents" section • Minor updates throughout chapter

Date	Version	Changes
November 2009	9.1.0	• Added details to “Creating Design Specifications” on page 1–2 • Added details to “Intellectual Property Selection” on page 1–2 • Updated information on “Device Selection” on page 1–3 • Added reference to “Device Migration Planning” on page 1–4 • Removed information from “Planning for Device Programming or Configuration” on page 1–4 • Added details to “Early Power Estimation” on page 1–5 • Updated information on “Early Pin Planning and I/O Analysis” on page 1–6 • Updated information on “Creating a Top-Level Design File for I/O Analysis” on page 1–8 • Added new “Simultaneous Switching Noise Analysis” section • Updated information on “Synthesis Tools” on page 1–9 • Updated information on “Simulation Tools” on page 1–9 • Updated information on “Planning for On-Chip Debugging Options” on page 1–10 • Added new “Managing Metastability” section • Changed heading title “Top-Down Versus Bottom-Up Incremental Flows” to “Single-Project Versus Multiple-Project Incremental Flows” • Updated information on “Creating a Design Floorplan” on page 1–18 • Removed information from “Fast Synthesis and Early Timing Estimation” on page 1–18
March 2009	9.0.0	• No change to content
November 2008	8.1.0	• Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	• Organization changes • Added “Creating Design Specifications” section • Added reference to new details in the In-System Design Debugging section of volume 3 • Added more details to the “Design Practices and HDL Coding Styles” section • Added references to the new Best Practices for Incremental Compilation and Floorplan Assignments chapter • Added reference to the Quartus II Language Templates

Related Information

Quartus II Handbook Archive

For previous versions of the Quartus II Handbook

Quartus II Incremental Compilation for Hierarchical and Team-Based Design 3

2015.05.04

QII5V1 Subscribe Send Feedback

About Quartus II Incremental Compilation

This manual provides information and design scenarios to help you partition your design to take advantage of the Quartus® II incremental compilation feature.

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

- Preserves the results and performance for unchanged logic in your design as you make changes elsewhere.
- Reduces design iteration time by an average of 75% for small changes in large designs, so that you can perform more design iterations per day and achieve timing closure efficiently.
- Facilitates modular hierarchical and team-based design flows, as well as design reuse and intellectual property (IP) delivery.

Quartus II incremental compilation supports the Arria®, Stratix®, and Cyclone® series of devices.

Deciding Whether to Use an Incremental Compilation Flow

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

Flat Compilation Flow with No Design Partitions

In the flat compilation flow with no design partitions, all the source code is processed and mapped during the Analysis and Synthesis stage, and placed and routed during the Fitter stage whenever the design is recompiled after a change in any part of the design. One reason for this behavior is to ensure optimal push-button quality of results. By processing the entire design, the Compiler can perform global optimizations to improve area and performance.

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

Related Information

Reducing Compilation Time documentation

Incremental Capabilities Available When A Design Has No Partitions

The Quartus II software has incremental compilation features available even when you do not partition your design, including Smart Compilation, Rapid Recompile, and incremental debugging. These features work in either an incremental or flat compilation flow.

With Smart Compilation

In any Quartus II compilation flow, you can use Smart Compilation to allow the Compiler to determine which compilation stages are required, based on the changes made to the design since the last smart compilation, and then skip any stages that are not required.
For example, when Smart Compilation is turned on, the Compiler skips the Analysis and Synthesis stage if all the design source files are unchanged. When Smart Compilation is turned on, if you make any changes to the logic of a design, the Compiler does not skip any compilation stage. You can turn on Smart Compilation on the **Compilation Process Settings** page of the **Setting** dialog box.

Note: Arria 10 devices do not support the smart compilation feature.

Related Information

Smart Compilation online help

With Rapid Recompile

The Quartus II software also includes a Rapid Recompile feature that instructs the Compiler to reuse the compatible compilation results if most of the design has not changed since the last compilation. This feature reduces compilation times for small and isolated design changes. You do not have control over which parts of the design are recompiled using this option; the Compiler determines which parts of the design must be recompiled. The Rapid Recompile feature preserves performance and can save compilation time by reducing the amount of changed logic that must be recompiled.

Related Information

About Rapid Recompile online help

With SignalTap II Logic Analyzer

During the debugging stage of the design cycle, you can add the SignalTap® II Logic Analyzer to your design, even if the design does not have partitions. To preserve the compilation netlist for the entire design, instruct the software to reuse the compilation results for the automatically-created "Top" partition that contains the entire design.

Related Information

About SignalTap II Logic Analyzer online help

Incremental Compilation Flow With Design Partitions

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

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

Incremental compilation is recommended for large designs and high resource densities when preserving results is important to achieve timing closure. The incremental compilation feature also facilitates team-based design flows that allow designers to create and optimize design blocks independently, when necessary.

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

In a team-based environment, part of your design may be incomplete, or it may have been developed by another designer or IP provider. In this scenario, you can add the completed partitions to the design incrementally. Alternatively, other designers or IP providers can develop and optimize partitions independently and the project lead can later integrate the partitions into the top-level design.

Related Information

- **Team-Based Design Flows and IP Delivery** on page 3-6
- **Incremental Compilation Summary** on page 3-7
- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Impact of Using Incremental Compilation with Design Partitions

Table 3-1: Impact Summary of Using Incremental Compilation

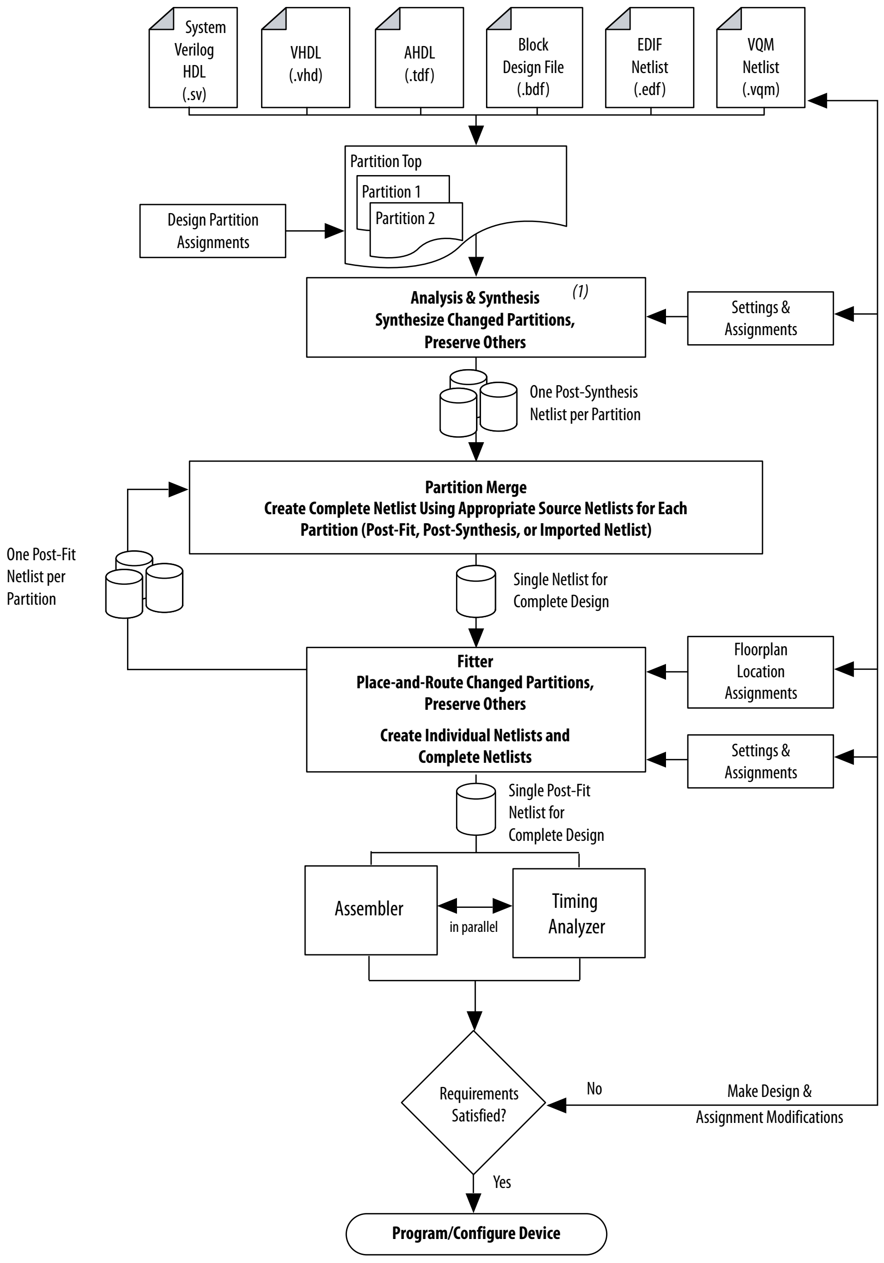
Characteristic	Impact of Incremental Compilation with Design Partitions
Compilation Time Savings	Typically saves an average of 75% of compilation time for small design changes in large designs when post-fit netlists are preserved; there are savings in both Quartus II Integrated Synthesis and the Fitter. (1)
Performance Preservation	Excellent performance preservation when timing critical paths are contained within a partition, because you can preserve post-fitting information for unchanged partitions.
Node Name Preservation	Preserves post-fitting node names for unchanged partitions.

(1) Quartus II incremental compilation does not reduce processing time for the early "pre-fitter" operations, such as determining pin locations and clock routing, so the feature cannot reduce compilation time if runtime is dominated by those operations.

Characteristic	Impact of Incremental Compilation with Design Partitions
Area Changes	The area (logic resource utilization) might increase because cross-boundary optimizations are limited, and placement and register packing are restricted.
f_{MAX} Changes	The design's maximum frequency might be reduced because cross-boundary optimizations are limited. If the design is partitioned and the floorplan location assignments are created appropriately, there might be no negative impact on f_{MAX}.

Quartus II Design Stages for Incremental Compilation

Figure 3-1: Design Stages for Incremental Compilation


System Verilog HDL (.sv)
VHDL (.vhd)
AHDL (.tdf)
Block Design File (.bdf)
EDIF Netlist (.edf)
VQM Netlist (.vqm)
Partition Top
Partition 1
Partition 2
Design Partition Assignments
Analysis & Synthesis (1)
Synthesize Changed Partitions, Preserve Others
Settings & Assignments
One Post-Synthesis Netlist per Partition
Partition Merge
Create Complete Netlist Using Appropriate Source Netlists for Each Partition (Post-Fit, Post-Synthesis, or Imported Netlist)
One Post-Fit Netlist per Partition
Single Netlist for Complete Design
Fitter
Place-and-Route Changed Partitions, Preserve Others
Create Individual Netlists and Complete Netlists
Floorplan Location Assignments
Settings & Assignments
Single Post-Fit Netlist for Complete Design
Assembler
in parallel
Timing Analyzer
Requirements Satisfied?
No
Make Design & Assignment Modifications
Yes
Program/Configure Device

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

Analysis and Synthesis Stage

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

Partition Merge Stage

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

Fitter Stage

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

How to Compare Incremental Compilation Results with Flat Design Results

If you define partitions, but want to check your compilation results without partitions in a “what if” scenario, you can direct the Compiler to ignore all partitions assignments in your project and compile the design as a "flat" netlist. When you turn on the **Ignore partitions assignments during compilation** option on the **Incremental Compilation** page, the Quartus II software disables all design partition assignments in your project and runs a full compilation ignoring all partition boundaries and netlists. Turning off the **Ignore partitions assignments during compilation** option restores all partition assignments and netlists for subsequent compilations.

Related Information

- **Incremental Compilation Page online help**
- **Design Partition Properties Dialog Box online help**

Team-Based Design Flows and IP Delivery

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

With a Single Quartus II Project

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

With Multiple Quartus II Projects

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

Additional Planning Needed

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

- Achieving timing closure for the full design may be more difficult if you compile partitions independently without information about other partitions in the design. This problem may be avoided by careful timing budgeting and special design rules, such as always registering the ports at the module boundaries.
- Resource budgeting and allocation may be required to avoid resource conflicts and overuse. Creating a floorplan with LogicLock regions is recommended when design partitions are developed independently in separate Quartus II projects.
- Maintaining consistency of assignments and timing constraints can be more difficult if there are separate Quartus II projects. The project lead must ensure that the top-level design and the separate projects are consistent in their assignments.

Collaboration on a Team-Based Design

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

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

Related Information

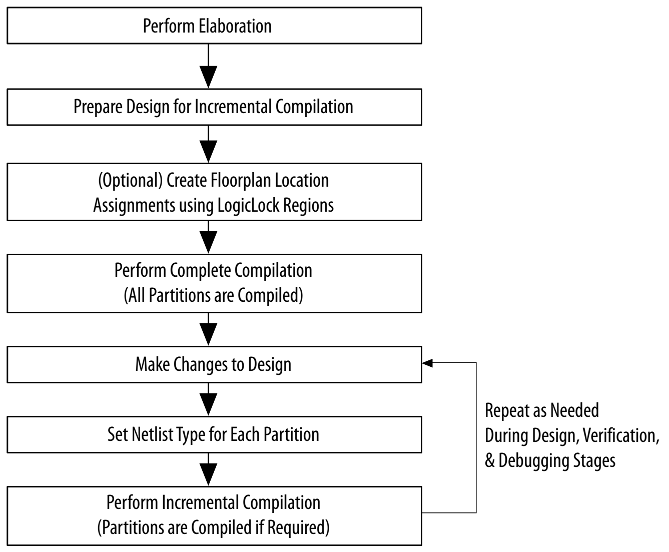
- **Exporting Design Partitions from Separate Quartus II Projects** on page 3-31
- **Project Management— Making the Top-Level Design Available to Other Designers** on page 3-33

Incremental Compilation Summary

Incremental Compilation Single Quartus II Project Flow

The figure illustrates the incremental compilation design flow when all partitions are contained in one top-level design.

Figure 3-2: Top-Down Design Flow


Perform Elaboration
Prepare Design for Incremental Compilation
(Optional) Create Floorplan Location Assignments using LogicLock Regions
Perform Complete Compilation (All Partitions are Compiled)
Make Changes to Design
Set Netlist Type for Each Partition
Perform Incremental Compilation (Partitions are Compiled if Required)
Repeat as Needed During Design, Verification, & Debugging Stages

Steps for Incremental Compilation

For an interactive introduction to implementing an incremental compilation design flow, refer to the **Getting Started Tutorial** on the Help menu in the Quartus II software.

Related Information

Using the Incremental Compilation Design Flow online help

Preparing a Design for Incremental Compilation

1. Elaborate your design, or run any compilation flow (such as a full compilation) that includes the elaboration step. Elaboration is the part of the synthesis process that identifies your design's hierarchy.
2. Designate specific instances in the design hierarchy as design partitions.
3. If required for your design flow, create a floorplan with LogicLock regions location assignments for timing-critical partitions that change with future compilations. Assigning a partition to a physical region on the device can help maintain quality of results and avoid conflicts in certain situations.

Related Information

- **Creating Design Partitions** on page 3-9
- **Creating a Design Floorplan With LogicLock Regions** on page 3-47

Compiling a Design Using Incremental Compilation

The first compilation after making partition assignments is a full compilation, and prepares the design for subsequent incremental compilations. In subsequent compilations of your design, you can preserve satisfactory compilation results and performance of unchanged partitions with the **Netlist Type** setting in the Design Partitions window. The **Netlist Type** setting determines which type of netlist or source file the Partition Merge stage uses in the next incremental compilation. You can choose the Source File, Post-Synthesis netlist, or Post-Fit netlist.

Related Information

Specifying the Level of Results Preservation for Subsequent Compilations on page 3-25

Creating Design Partitions

There are several ways to designate a design instance as a design partition.

Related Information

Deciding Which Design Blocks Should Be Design Partitions on page 3-20

Creating Design Partitions in the Project Navigator

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

Related Information

Creating Design Partitions online help

Creating Design Partitions in the Design Partitions Window

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

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

You can right-click a partition in the window to perform various common tasks, such as viewing property information about a partition, including the time and date of the compilation netlists and the partition statistics.

When you create a partition, the Quartus II software automatically generates a name based on the instance name and hierarchy path. You can edit the partition name in the Design Partitions Window so that you avoid referring to them by their hierarchy path, which can sometimes be long. This is especially useful when using command-line commands or assignments, or when you merge partitions to give the partition a meaningful name. Partition names can be from 1 to 1024 characters in length and must be unique. The name can consist of alphanumeric characters and the pipe (|), colon (:), and underscore (_) characters.

Related Information

- Netlist Type for Design Partitions on page 3-25
- Creating Design Partitions online help

Creating Design Partitions With the Design Partition Planner

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

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

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

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

Related Information

Using the Design Partition Planner online help

Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation on page 14-1

Creating Design Partitions With Tcl Scripting

You can also create partitions with Tcl scripting commands.

Related Information

Scripting Support on page 3-54

Automatically-Generated Partitions

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

Common Design Scenarios Using Incremental Compilation

Related Information

Steps for Incremental Compilation on page 3-8

Reducing Compilation Time When Changing Source Files for One Partition

Scenario background: You set up your design to include partitions for several of the major design blocks, and now you have just performed a lengthy compilation of the entire design. An error is found in the HDL source file for one partition and it is being fixed. Because the design is currently meeting timing requirements, and the fix is not expected to affect timing performance, it makes sense to compile only the affected partition and preserve the rest of the design.

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

Perform the following steps to update a single source file:

1. Apply and save the fix to the HDL source file.
2. On the Assignments menu, open the **Design Partitions window**.
3. Change the netlist type of each partition, including the top-level entity, to **Post-Fit** to preserve as much as possible for the next compilation.

- The Quartus II software recompiles partitions by default when changes are detected in a source file. You can refer to the Partition Dependent Files table in the Analysis and Synthesis report to determine which partitions were recompiled. If you change an assignment but do not change the logic in a source file, you can set the netlist type to **Source File** for that partition to instruct the software to recompile the partition's source design files and its assignments.

4. Click **Start Compilation** to incrementally compile the fixed HDL code. This compilation should take much less time than the initial full compilation.
5. Simulate the design to ensure that the error is fixed, and use the TimeQuest Timing Analyzer report to ensure that timing results have not degraded.

Related Information

List of Compilation and Simulation Reports online help

Optimizing a Timing-Critical Partition

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

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

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

1. Open the **Design Partitions** window.
2. For the partition in question, set the netlist type to **Source File**.
   - If you change a setting that affects only the Fitter, you can save additional compilation time by setting the netlist type to **Post-Synthesis** to reuse the synthesis results and refit the partition.
3. For the remaining partitions (including the top-level entity), set the netlist type to **Post-Fit**.
   - You can optionally set the **Fitter Preservation Level** on the **Advanced** tab in the **Design Partitions Properties** dialog box to **Placement** to allow for the most flexibility during routing.
4. Apply the desired optimization settings.
5. Click **Start Compilation** to perform incremental compilation on the design with the new settings. During this compilation, the Partition Merge stage automatically merges the critical partition's new synthesis netlist with the post-fit netlists of the remaining partitions. The Fitter then refits only the required partition. Because the effort is reduced as compared to the initial full compilation, the compilation time is also reduced.

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

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

Adding Design Logic Incrementally or Working With an Incomplete Design

Scenario background: You have one or more partitions that are known to be timing-critical in your full design. You want to focus on developing and optimizing this subset of the design first, before adding the rest of the design logic.

Use this flow to compile a timing-critical partition or partitions in isolation, optionally with extra optimizations turned on. After timing closure is achieved for the critical logic, you can preserve its content and placement and compile the remaining partitions with normal or reduced optimization levels. For example, you may want to compile an IP block that comes with instructions to perform optimization before you incorporate the rest of your custom logic.

To implement this design flow, perform the following steps:

1. Partition the design and create floorplan location assignments. For best results, ensure that the top-level design includes the entire project framework, even if some parts of the design are incomplete and are represented by an empty wrapper file.
2. For the partitions to be compiled first, in the Design Partitions window, set the netlist type to **Source File**.
3. For the remaining partitions, set the netlist type to **Empty**.
4. To compile with the desired optimizations turned on, click **Start Compilation**.
5. Check the Timing Analyzer reports to ensure that timing requirements are met. If so, proceed to step 6. Otherwise, repeat steps 4 and 5 until the requirements are met.
6. In the Design Partitions window, set the netlist type to **Post-Fit** for the first partitions. You can set the **Fitter Preservation Level** on the **Advanced** tab in the **Design Partitions Properties** dialog box to **Placement** to allow more flexibility during routing if exact placement and routing preservation is not required.
7. Change the netlist type from **Empty** to **Source File** for the remaining partitions, and ensure that the completed source files are added to the project.
8. Set the appropriate level of optimizations and compile the design. Changing the optimizations at this point does not affect any fitted partitions, because each partition has its netlist type set to **Post-Fit**.
9. Check the Timing Analyzer reports to ensure that timing requirements are met. If not, make design or option changes and repeat step 8 and step 9 until the requirements are met.

The flow in this example is similar to design flows in which a module is implemented separately and is later merged into the top-level. Generally, optimization in this flow works only if each critical path is contained within a single partition. Ensure that if there are any partitions representing a design file that is missing from the project, you create a placeholder wrapper file to define the port interface.

Related Information

- **Designing in a Team-Based Environment** on page 3-41
- **Deciding Which Design Blocks Should Be Design Partitions** on page 3-20
- **Empty Partitions** on page 3-33

Debugging Incrementally With the SignalTap II Logic Analyzer

Scenario background: Your design is not functioning as expected, and you want to debug the design using the SignalTap II Logic Analyzer. To maintain reduced compilation times and to ensure that you do not negatively affect the current version of your design, you want to preserve the synthesis and fitting results and add the SignalTap II Logic Analyzer to your design without recompiling the source code.

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

It is not necessary to create design partitions in order to use the SignalTap II incremental compilation feature. The SignalTap II Logic Analyzer acts as its own separate design partition.

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

1. Open the Design Partitions window.
2. Set the netlist type to **Post-fit** for all partitions to preserve their placement.
    - The netlist type for the top-level partition defaults to **Source File**, so be sure to change this "Top" partition in addition to any design partitions that you have created.
3. If you have not already compiled the design with the current set of partitions, perform a full compilation. If the design has already been compiled with the current set of partitions, the design is ready to add the SignalTap II Logic Analyzer.
4. Set up your SignalTap II File using the **post-fitting** filter in the **Node Finder** to add signals for logic analysis. This allows the Fitter to add the SignalTap II logic to the post-fit netlist without modifying the design results.

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

Related Information

Design Debugging Using the SignalTap II Embedded Logic Analyzer documentation

Functional Safety IP Implementation

In functional safety designs, recertification is required when logic is modified in safety or standard areas of the design. Recertification is required because the FPGA programming file has changed. You can reduce the amount of required recertification if you use the functional safety separation flow in the Quartus II software. By partitioning your safety IP (SIP) from standard logic, you ensure that the safety critical areas of the design remain the same when the standard areas in your design are modified. The safety-critical areas remain the same at the bit level.

The functional safety separation flow supports only Cyclone IV and Cyclone V device families.

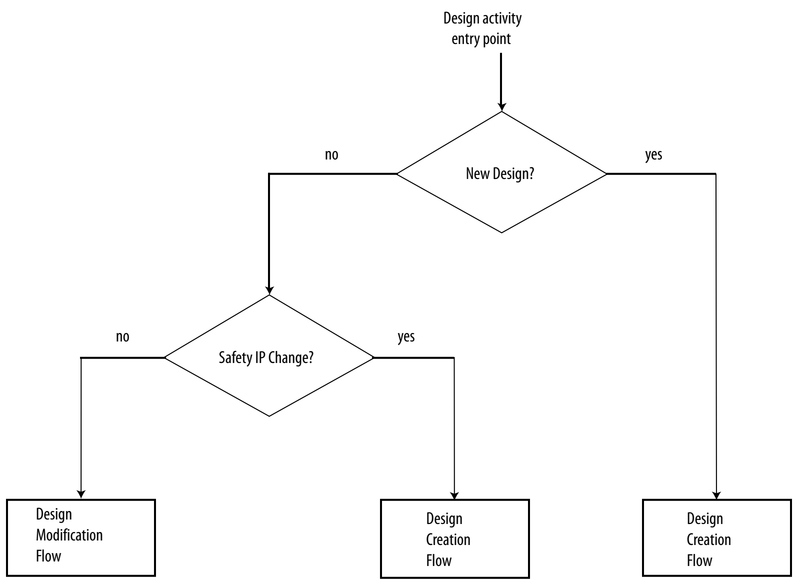
Software Tool Impact on Safety

The Quartus II software can partition your design into safety partitions and standard partitions, but the Quartus II software does not perform any online safety-related functionality. The Quartus II software generates a bitstream that performs the safety functions. For the purpose of compliance with a functional safety standard, the Quartus II software should be considered as an offline support tool.

Functional Safety Separation Flow

The functional safety separation flow consists of two separate work flows. The design creation flow and the design modification flow both use incremental compilation, but the two flows have different use-case scenarios.

Figure 3-3: Functional Safety Separation Flow

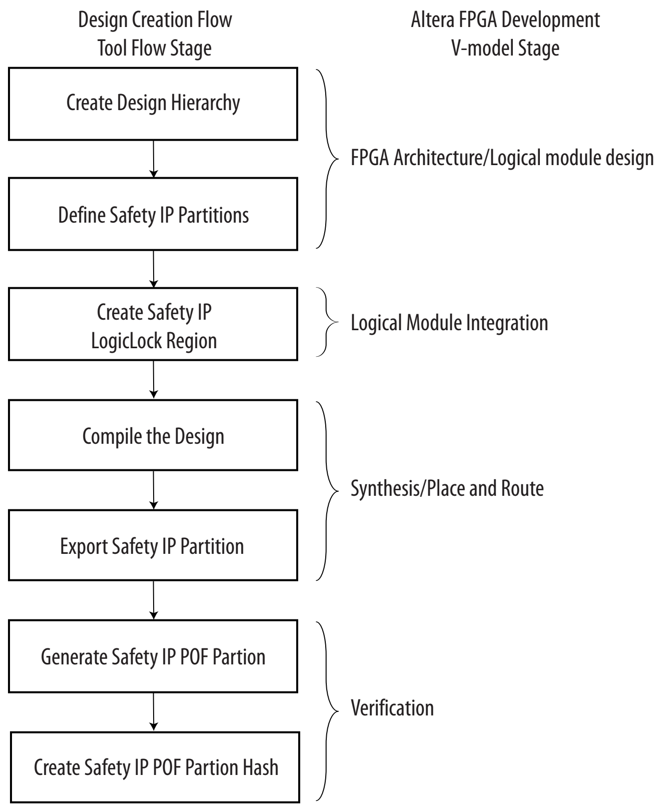


Design Creation Flow

The design creation flow describes the necessary steps for initial design creation in a way that allows you to modify your design. Some of the steps are architectural constraints and the remaining steps are steps that you need to perform in the Quartus II software. Use the design creation flow for the first pass certification of your product.

When you make modifications to the safety IP in your design, you must use the design creation flow.

Figure 3-4: Design Creation Flow


Design Creation Flow Tool Flow Stage
Altera FPGA Development V-model Stage
Create Design Hierarchy
Define Safety IP Partitions
FPGA Architecture/Logical module design
Create Safety IP LogicLock Region
Logical Module Integration
Compile the Design
Export Safety IP Partition
Synthesis/Place and Route
Generate Safety IP POF Partion
Create Safety IP POF Partion Hash
Verification

The design creation flow becomes active when you have a valid safety IP partition in your Quartus II project and that safety IP partition does not have place and route data from a previous compile. In the design creation flow, the Assembler generates a Partial Settings Mask (**.psm**) file for each safety IP partition. Each **.psm** file contains a list of programming bits for its respective safety IP partition.

The Quartus II software determines whether to use the design creation flow or design modification flow on a per partition basis. It is possible to have multiple safety IP partitions in a design where some are running the design creation flow and others are running the design modification flow.

To reset the complete design to the design creation flow, remove the previous place and route data by cleaning the project (removing the dbs). Alternatively, use the partition import flow, to selectively reset the design. You can remove the netlists for the imported safety IP partitions individually using the **Design Partitions** window.

Related Information

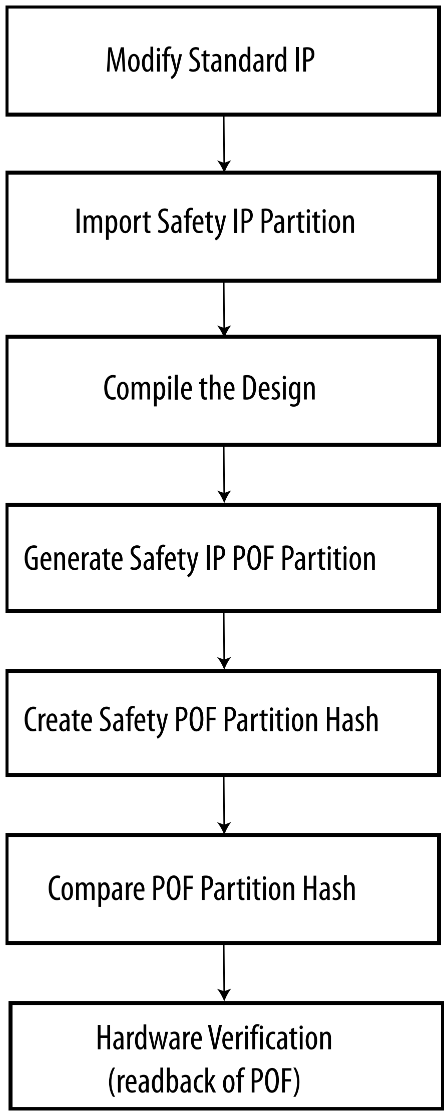
- **Exporting and Importing Your Safety IP** on page 3-19
- **Design Partitions Window online help**

Design Modification Flow

The design modification flow describes the necessary steps to make modifications to the standard IP in your design. This flow ensures that the previously compiled safety IP that the project uses remains unchanged when you change or compile standard IP.

Use the design modification flow only after you qualify your design in the design creation flow.

Figure 3-5: Design Modification Flow


Modify Standard IP
Import Safety IP Partition
Compile the Design
Generate Safety IP POF Partition
Create Safety POF Partition Hash
Compare POF Partition Hash
Hardware Verification (readback of POF)

When the design modification flow is active for a safety IP partition, the Fitter runs in Strict Preservation mode for that partition. The Assembler performs run-time checks that compare the Partial Settings Mask information matches the **.psm** file generated in the design creation flow. If the Assembler detects a mismatch, a "Bad Mask!" or "ASM_STRICT_PRESERVA-
TION_BITS_UTILITY::compare_masked_byte_array failed" internal error message is shown. If you see either error message while compiling your design, contact **Altera support** for assistance.

When a change is made to any HDL source file that belongs to a safety IP, the default behavior of the Quartus II software is to resynthesize and perform a clean place and route for that partition, which then activates the design creation flow for that partition. To change this default behavior and keep the design modification flow active, do the following:

- Use the partition export/import flow.

or

- Use the Design Partitions window to modify the design partition properties and turn on **Ignore changes in source files and strictly use the specified netlist, if available**.

The Fitter applies the same design flow to all partitions that belong to the same safety IP. If more than one safety IP is used in the design, the Fitter may evoke different flows for different safety IPs.

Note: If your safety IP is a sub-block in a Qsys system, every time you regenerate HDL for the Qsys system, the timestamp for the safety IP HDL changes. This results in resynthesis of the safety IP, unless the default behavior (described above) is changed.

Related Information

- **Exporting and Importing Your Safety IP** on page 3-19
- **Design Partitions Window online help**

How to Turn On the Functional Safety Separation Flow

Every safety-related IP component in your design should be implemented in a partition(s) so the safety IPs are protected from recompilation. The global assignment `PARTITION_ENABLE_STRICT_PRESERVATION` is used to identify safety IP in your design.

```
set_global_assignment -name PARTITION_ENABLE_STRICT_PRESERVATION <ON/OFF> -
section_id <partition_name>
```

When this global assignment is designated as ON for a partition, the partition is protected from recompilation, exported as a safety IP, and included in the safety IP POF mask. Specifying the value as ON for any partition turns on the functional safety separation flow.

When this global assignment is designated as OFF, the partition is considered as standard IP or as not having a `PARTITION_ENABLE_STRICT_PRESERVATION` assignment at all. Logic that is not assigned to a partition is considered as part of the top partition and treated as standard logic.

Note: Only partitions and I/O pins can be assigned to SIP.

A partition assigned to safety IP can contain safety logic only. If the parent partition is assigned to a safety IP, then all the child partitions for this parent partition are considered as part of the safety IP. If you do not explicitly specify a child partition as a safety IP, a critical warning notifies you that the child partition is treated as part of a safety IP.

A design can contain several safety IPs. All the partitions containing logic that implements a single safety IP function should belong with the same top-level parent partition.

You can also turn on the functional safety separation flow from the **Design Partition Properties** dialog box. Click the **Advanced** tab and turn on **Allow partition to be strictly preserved for safety**.

When the functional safety separation flow is active, you can view which partitions in your design have the Strict Preservation property turned on. The **Design Partitions** window displays a on or off value for safety IP in your design (in the **Strict Preservation** column).

Related Information

- **Design Partition Properties Dialog box online help**
- **Design Partitions Window online help**

Preservation of Device Resources

The preservation of the partition’s netlist atoms and the atoms placement and routing, in the design modification flow, is done by setting the netlist type to **Post-fit** with the Fitter preservation level set to **Placement and Routing Preserved**.

Preservation of Placement in the Device with LogicLock

In order to fix the safety IP logic into specific areas of the device, you should define LogicLock regions. By using preserved LogicLock regions, device placement is reserved for the safety IP to prevent standard logic from being placed into the unused resources of the safety IP region. You establish a fixed size and origin to ensure location preservation. You need to use LogicLock to ensure a valid safety IP POF mask is generated when you turn on the functional safety separation flow. The POF comparison tool for functional safety can check that the safety region is unchanged between compiles. A LogicLock region assigned to a safety IP can only contain safety IP logic.

Assigning I/O Pins

You can use a global assignment to specify that a pin is assigned to a safety IP.

```
set_instance_assignment -name ENABLE_STRICT_PRESERVATION ON/OFF -to <hpath> -
section_id <region_name>
```

- <*hpath*> refers to an I/O pin (pad).
- <*region_name*> refers to the top-level safety IP partition name.

A value of ON indicates that the pin is a safety pin that should be preserved along with the safety IP. A value of OFF indicates that the pin that connects up to the safety IP, should be treated as a standard pin, and is not preserved along with the safety IP.

All the pins that connect up to a safety IP should have an explicit assignment.

An error is reported if a pin that connects up the safety IP does not have an assignment or a pin does not connect up to the specified <*region_name*>.

If an IO_REG group contains a pin that is assigned to a safety IP, then all the pins in the IO_REG group are reserved for this safety IP. All pins in the IO_REG group need to be assigned to the same safety IP and none of the pins in the group can be assigned to standard signals.

General Guidelines for Implementation

- An internal clock source, such as a PLL, should be implemented in a safe partition.
- An I/O pin driving the external clock should be indicated as a safety pin.
- To export a safety IP containing several partitions, the top-level partition for the safety IP should be exported. A safety IP containing several partitions is flattened and converted into a single partition during export. This hierarchical safety IP is flattened to enure bit-level settings are preserved.
- Hard blocks implemented in a safe partition needs to stay with the safe partition.

Reports for Safety IP

When you have the functional safety separation flow turned on, the Quartus II software displays safety IP and standard IP information in the Fitter report.

Fitter Report

The Fitter report includes information for each safety IP and the respective partition and I/O usage. The report contains the following information:

- Safety IP name defined as the name of the top-level safety IP partition
- Effective design flow for the safety IP
- Names of all partitions that belong to the safety IP
- Number of safety/standard inputs to the safety IP
- Number of safety/standard outputs to the safety IP
- LogicLock region names along with size and locations for the regions
- I/O pins used for the respective safety IP in your design
- Safety-related error messages

SIP Partial Bitstream Generation

The Programmer generates a bitstream file containing only the bits for a safety IP. This partial preserved bitstream (**.ppb**) file is for the safety IP region mask. The command lines to generate the partial bitstream file are the following:

- `quartus_cpf --genppb safe1.psm design.sof safe1.rbf.ppb`
- `quartus_cpf -c safe1.psm safe1.rbf.ppb`

The **.ppb** file is generated in two steps.

1. Generation of partial SOF.
2. Generation of **.ppb** file using the partial SOF.

The **.psm** file, **.ppb** file, and MD5 hash signature (**.md5.sign**) file created during partial bitstream generation should be archived for use in future design modification flow compiles.

Exporting and Importing Your Safety IP

Safety IP Partition Export

After you have successfully compiled the safety IP(s) in the Quartus II software, save the safety partition place and route information for use in any subsequent design modification flow. Saving the partition information allows the safety IP to be imported to a clean Quartus II project where no previous compilation results have been removed (even if the version of the Quartus II software being used is newer than the Quartus II software version with which the safety IP was originally compiled). Use the **Design Partitions** window to export the design partition. Verify that only the post-fit netlist and export routing options are turned on when you generate the **.qxp** file for each safety IP. The **.qxp** files should be archived along with the partial bitstream files for use in later design modification flow compiles.

Safety IP Partition Import

You can import a previously exported safety IP partition into your Quartus II project. There are two use-cases for this.

- (Optional) Import into the original project to ensure that any potential source code changes do not trigger the design creation flow unintentionally.
- Import into a new or clean project where you want to use the design modification flow for the safety IP. As the exported partition is independent of your Quartus II software version, you can import the **.qxp** into a future Quartus II software release.

To import a previously exported design partition, use the **Design Partitions** window and import the **.qxp**.

Related Information

- Export Design Partition online help
- Import Design Partition online help

POF Comparison Tool for Verification

There is a separate safe/standard partitioning verification tool that is licensed to safety users. Along with the **.ppb** file, a **.md5.sign** file is generated. The MD5 hash signature can be used for verification. For more detailed verification, the POF comparison tool should be used. This POF comparison tool is available in the Altera Functional Safety Data Package.

Deciding Which Design Blocks Should Be Design Partitions

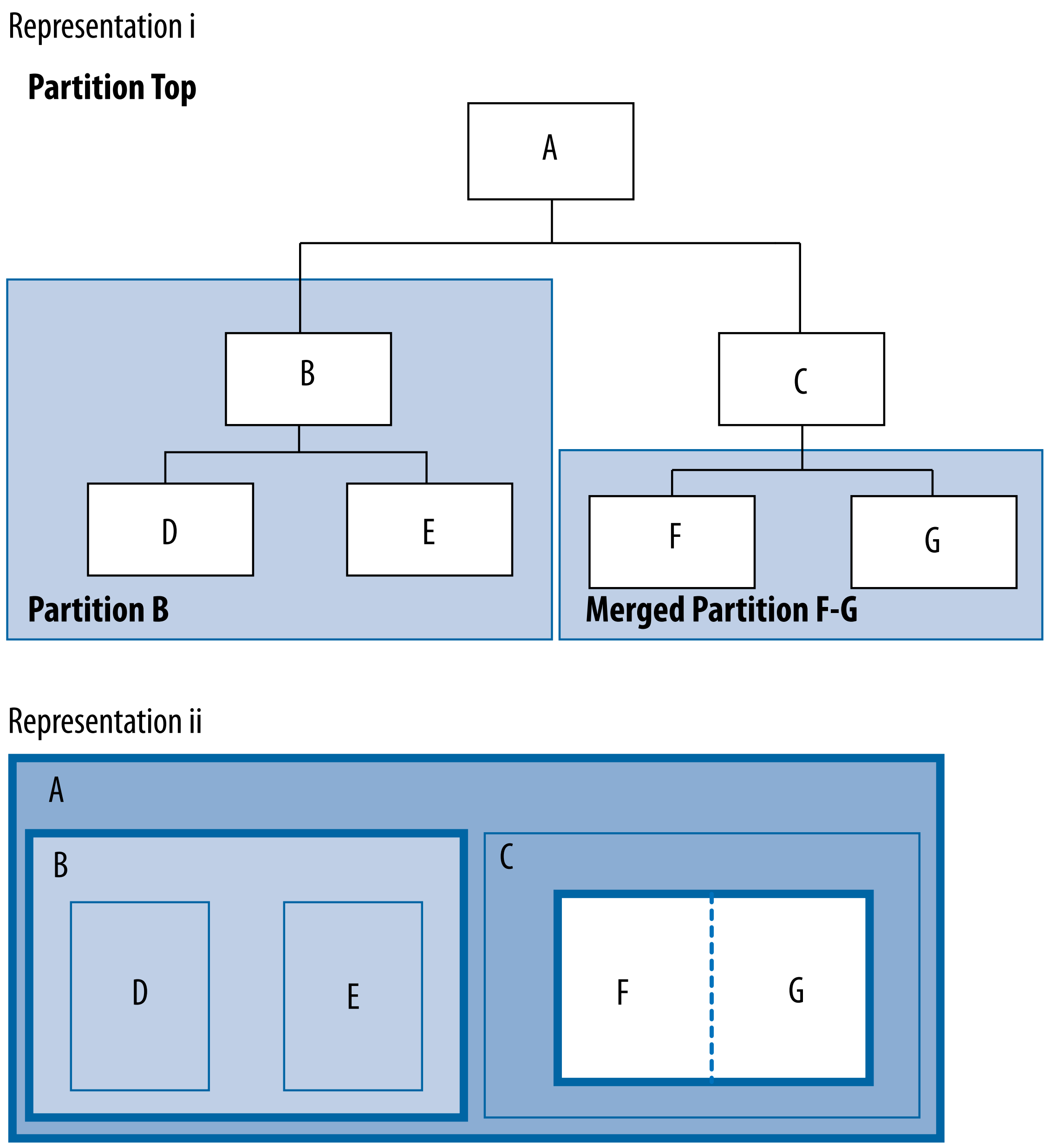
The incremental compilation design flow requires more planning than flat compilations. For example, you might have to structure your source code or design hierarchy to ensure that logic is grouped correctly for optimization.

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

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

In the figure below, a complete design is made up of instances **A**, **B**, **C**, **D**, **E**, **F**, and **G**. The shaded boxes in Representation i indicate design partitions in a "tree" representation of the hierarchy. In Representation ii, the lower-level instances are represented inside the higher-level instances, and the partitions are illustrated with different colored shading. The top-level partition, called "Top", automatically contains the top-level entity in the design, and contains any logic not defined as part of another partition. The design file for the top level may be just a wrapper for the hierarchical instances below it, or it may contain its own logic. In this example, partition **B** contains the logic in instances **B**, **D**, and **E**. Entities **F** and **G** were first identified as separate partitions, and then merged together to create a partition **F**-**G**. The partition for the top-level entity **A**, called "Top", includes the logic in one of its lower-level instances, **C**, because **C** was not defined as part of any other partition.

Figure 3-6: Partitions in a Hierarchical Design



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

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

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

Impact of Design Partitions on Design Optimization

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

Whenever possible, register all inputs and outputs of each partition. This helps avoid any delay penalty on signals that cross partition boundaries and keeps each register-to-register timing path within one partition for optimization. In addition, minimize the number of paths that cross partition boundaries. If there are timing-critical paths that cross partition boundaries, rework the partitions to avoid these inter-partition paths. Including as many of the timing-critical connections as possible inside a partition allows you to effectively apply optimizations to that partition to improve timing, while leaving the rest of the design unchanged.

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

If critical timing paths cross partition boundaries, you can perform timing budgeting and make timing assignments to constrain the logic in each partition so that the entire timing path meets its requirements. In addition, because each partition is optimized independently during synthesis, you may have to perform resource allocation to ensure that each partition uses an appropriate number of device resources. If design partitions are compiled in separate Quartus II projects, there may be conflicts related to global routing resources for clock signals when the design is integrated into the top-level design. You can use the Global Signal logic option to specify which clocks should use global or regional routing, use the ALTCLK_CTRL IP core to instantiate a clock control block and connect it appropriately in both the partitions being developed in separate Quartus II projects, or find the compiler-generated clock control node in your design and make clock control location assignments in the Assignment Editor.

Turning On Supported Cross-boundary Optimizations

You can improve the optimizations performed between design partitions by turning on supported cross-boundary optimizations. These optimizations are turned on a per partition basis and you can select the optimizations as individual assignments. This allows the cross-boundary optimization feature to give you more control over the optimizations that work best for your design. You can turn on the cross-boundary optimizations for your design partitions on the **Advanced** tab of the **Design Partition Properties** dialog box. Once you change the optimization settings, the Quartus II software recompiles your partition from source automatically. Cross-boundary optimizations include the following: propagate constants, propagate inversions on partition inputs, merge inputs fed by a common source, merge electrically equivalent bidirectional pins, absorb internal paths, and remove logic connected to dangling outputs.

Cross-boundary optimizations are implemented top-down from the parent partition into the child partition, but not vice-versa. Also, cross-boundary optimizations cannot be enabled for partitions that allow multiple personas (partial reconfiguration partitions).

Related Information

Design Partition Properties Dialog Box online help

Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation on page 14-1

Design Partition Assignments Compared to Physical Placement Assignments

Design partitions for incremental compilation are logical partitions, which is different from physical placement assignments in the device floorplan. A logical design partition does not refer to a physical area of the device and does not directly control the placement of instances. A logical design partition sets up a virtual boundary between design hierarchies so that each is compiled separately, preventing logical optimizations from occurring between them. When the software compiles the design source code, the logic in each partition can be placed anywhere in the device unless you make additional placement assignments.

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

You can assign design partitions to physical regions in the device floorplan using LogicLock region assignments. In the Quartus II software, LogicLock regions are used to constrain blocks of a design to a particular region of the device. Altera recommends using LogicLock regions for timing-critical design blocks that will change in subsequent compilations, or to improve the quality of results and avoid placement conflicts in some cases.

Related Information

Creating a Design Floorplan With LogicLock Regions on page 3-47

Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation on page 14-1

Using Partitions With Third-Party Synthesis Tools

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

Synopsys Synplify Pro/Premier and Mentor Graphics Precision RTL Plus

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

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

Related Information

- **Synopsys Synplify Support documentation** on page 17-1
- **Mentor Graphics Precision Synthesis Support documentation** on page 18-1

Other Synthesis Tools

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

Assessing Partition Quality

The Quartus II software provides various tools to assess the quality of your assigned design partitions. You can take advantage of these tools to assess your partition quality, and use the information to improve your design or assignments as required to achieve the best results.

Partition Statistics Reports

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

The Partition Merge Partition Statistics report lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells it contains, as well as the number of input and output pins it contains, and how many are registered or unconnected.

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

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Partition Timing Reports

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

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

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

Incremental Compilation Advisor

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

Recommendations are split into **General Recommendations**, **Timing Recommendations**, and **Team-Based Design Recommendations** that apply to design flows in which partitions are compiled independently in separate Quartus II projects before being integrated into the top-level design. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make a suggested change. In some cases, there is a link to the appropriate Quartus II settings page where you can make a suggested change to assignments or settings. For some items, if your design does not follow the recommendation, the **Check Recommendations** operation creates a table that lists any nodes or paths in your design that could be improved. The relevant timing-independent recommendations for the design are also listed in the Design Partitions window and the LogicLock Regions window.

To verify that your design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page, and then click **Check Recommendations**. For large designs, these operations can take a few minutes.

After you perform a check operation, symbols appear next to each recommendation to indicate whether the design or project setting follows the recommendations, or if some or all of the design or project settings do not follow the recommendations. Following these recommendations is not mandatory to use the incremental compilation feature. The recommendations are most important to ensure good results for timing-critical partitions.

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

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

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

Specifying the Level of Results Preservation for Subsequent Compilations

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

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

Netlist Type for Design Partitions

Before starting a new compilation, ensure that the appropriate netlist type is set for each partition to preserve the desired level of compilation results. The table below describes the settings for the netlist type, explains the behavior of the Quartus II software for each setting, and provides guidance on when to use each setting.

Table 3-2: Partition Netlist Type Settings

Netlist Type	Quartus II Software Behavior for Partition During Compilation
Source File	Always compiles the partition using the associated design source file(s). [2] Use this netlist type to recompile a partition from the source code using new synthesis or Fitter settings.
Post-Synthesis	Preserves post-synthesis results for the partition and reuses the post-synthesis netlist when the following conditions are true: • A post-synthesis netlist is available from a previous synthesis. • No change that initiates an automatic resynthesis has been made to the partition since the previous synthesis. [3] Compiles the partition from the source files if resynthesis is initiated or if a post-synthesis netlist is not available. [2] Use this netlist type to preserve the synthesis results unless you make design changes, but allow the Fitter to refit the partition using any new Fitter settings.
Post-Fit	Preserves post-fit results for the partition and reuses the post-fit netlist when the following conditions are true: • A post-fit netlist is available from a previous fitting. • No change that initiates an automatic resynthesis has been made to the partition since the previous fitting. [3] When a post-fit netlist is not available, the software reuses the post-synthesis netlist if it is available, or otherwise compiles from the source files. Compiles the partition from the source files if resynthesis is initiated. [2] The Fitter Preservation Level specifies what level of information is preserved from the post-fit netlist. Assignment changes, such as Fitter optimization settings, do not cause a partition set to Post-Fit to recompile.

[2] If you use Rapid Recompile, the Quartus II software might not recompile the entire partition from the source code as described in this table; it will reuse compatible results if there have been only small changes to the logic in the partition.

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

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

Related Information

- **What Changes Initiate the Automatic Resynthesis of a Partition?** on page 3-29
- **Fitter Preservation Level for Design Partitions** on page 3-27
- **Incremental Capabilities Available When A Design Has No Partitions** on page 3-2

Fitter Preservation Level for Design Partitions

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

You can change the advanced Fitter Preservation Level setting to provide more flexibility in the Fitter during placement and routing. You can set the Fitter Preservation Level on the **Advanced** tab in the **Design Partitions Properties** dialog box.

Table 3-3: Fitter Preservation Level Settings

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Placement and Routing	Preserves the design partition's netlist atoms and their placement and routing. This setting reduces compilation times compared to Placement only, but provides less flexibility to the router to make changes if there are changes in other parts of the design. By default, the Fitter preserves the usage of high-speed programmable power tiles contained within the selected partition, for devices that support high-speed and low-power tiles. You can turn off the **Preserve high-speed tiles when preserving placement and routing** option on the **Advanced** tab in the **Design Partitions Properties** dialog box.
Placement	Preserves the netlist atoms and their placement in the design partition. Reroutes the design partition and does not preserve high-speed power tile usage.
Netlist Only	Preserves the netlist atoms of the design partition, but replaces and reroutes the design partition. A post-fit netlist with the atoms preserved can be different than the Post-Synthesis netlist because it contains Fitter optimizations; for example, Physical Synthesis changes made during a previous Fitting. You can use this setting to: • Preserve Fitter optimizations but allow the software to perform placement and routing again. • Reapply certain Fitter optimizations that would otherwise be impossible when the placement is locked down. • Resolve resource conflicts between two imported partitions.

Related Information

Setting the Netlist Type and Fitter Preservation Level for Design Partitions online help

Where Are the Netlist Databases Saved?

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

If you archive or reproduce the project in another location, you can use a Quartus II Archive File (**.qar**). Include the incremental compilation database files to preserve post-synthesis or post-fit compilation results.

To manually create a project archive that preserves compilation results without keeping the incremental compilation database, you can keep all source and settings files, and create and save a Quartus II Settings File (**.qxp**) for each partition in the design that will be integrated into the top-level design.

Related Information

- **Using Incremental Compilation With Quartus II Archive Files** on page 3-49
- **Exporting Design Partitions from Separate Quartus II Projects** on page 3-31

Deleting Netlists

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

What Changes Initiate the Automatic Resynthesis of a Partition?

A partition is synthesized from its source files if there is no post-synthesis netlist available from a previous synthesis, or if the netlist type is set to **Source File**. Additionally, certain changes to a partition initiate an automatic resynthesis of the partition when the netlist type is **Post Synthesis** or **Post-Fit**. The software resynthesizes the partition in these cases to ensure that the design description matches the post-place-and-route programming files.

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

- The device family setting has changed.
- Any dependent source design file has changed.
- The partition boundary was changed by an addition, removal, or change to the port boundaries of a partition (for example, a new partition has been defined for a lower-level instance within this partition).
- A dependent source file was compiled into a different library (so it has a different `-library` argument).
- A dependent source file was added or removed; that is, the partition depends on a different set of source files.

- The partition's root instance has a different entity binding. In VHDL, an instance may be bound to a specific entity and architecture. If the target entity or architecture changes, it triggers resynthesis.
- The partition has different parameters on its root hierarchy or on an internal AHDL hierarchy (AHDL automatically inherits parameters from its parent hierarchies). This occurs if you modified the parameters on the hierarchy directly, or if you modified them indirectly by changing the parameters in a parent design hierarchy.
- You have moved the project and compiled database between a Windows and Linux system. Due to the differences in the way new line feeds are handled between the operating systems, the internal checksum algorithm may detect a design file change in this case.

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

Synthesis and Fitter assignments, such as optimization settings, timing assignments, or Fitter location assignments including pin assignments, do not trigger automatic recompilation in the incremental compilation flow. To recompile a partition with new assignments, change the netlist type for that partition to one of the following:

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

You can use the LogicLock Origin location assignment to change or fine-tune the previous Fitter results from a Post-Fit netlist.

Related Information

Changing Partition Placement with LogicLock Changes on page 3-48

Resynthesis Due to Source Code Changes

The Quartus II software uses an internal checksum algorithm to determine whether the contents of a source file have changed. Source files are the design description files used to create the design, and include Memory Initialization Files (**.mif**) as well as **.qxp** from exported partitions. When design files in a partition have dependencies on other files, changing one file may initiate an automatic recompilation of another file. The Partition Dependent Files table in the Analysis and Synthesis report lists the design files that contribute to each design partition. You can use this table to determine which partitions are recompiled when a specific file is changed.

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

Note: If you use Rapid Recompile, the Quartus II software might not recompile the entire partition from the source code as described in this section; it will reuse compatible results if there have been only small changes to the logic in the partition.

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

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

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

Related Information

Incremental Capabilities Available When A Design Has No Partitions on page 3-2

Forcing Use of the Compilation Netlist When a Partition has Changed

Forcing the use of a post-compilation netlist when the contents of a source file has changed is recommended only for advanced users who understand when a partition must be recompiled. You might use this assignment, for example, if you are making source code changes but do not want to recompile the partition until you finish debugging a different partition, or if you are adding simple comments to the source file but you know the design logic itself is not being changed and you want to keep the previous compilation results.

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

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

Exporting Design Partitions from Separate Quartus II Projects

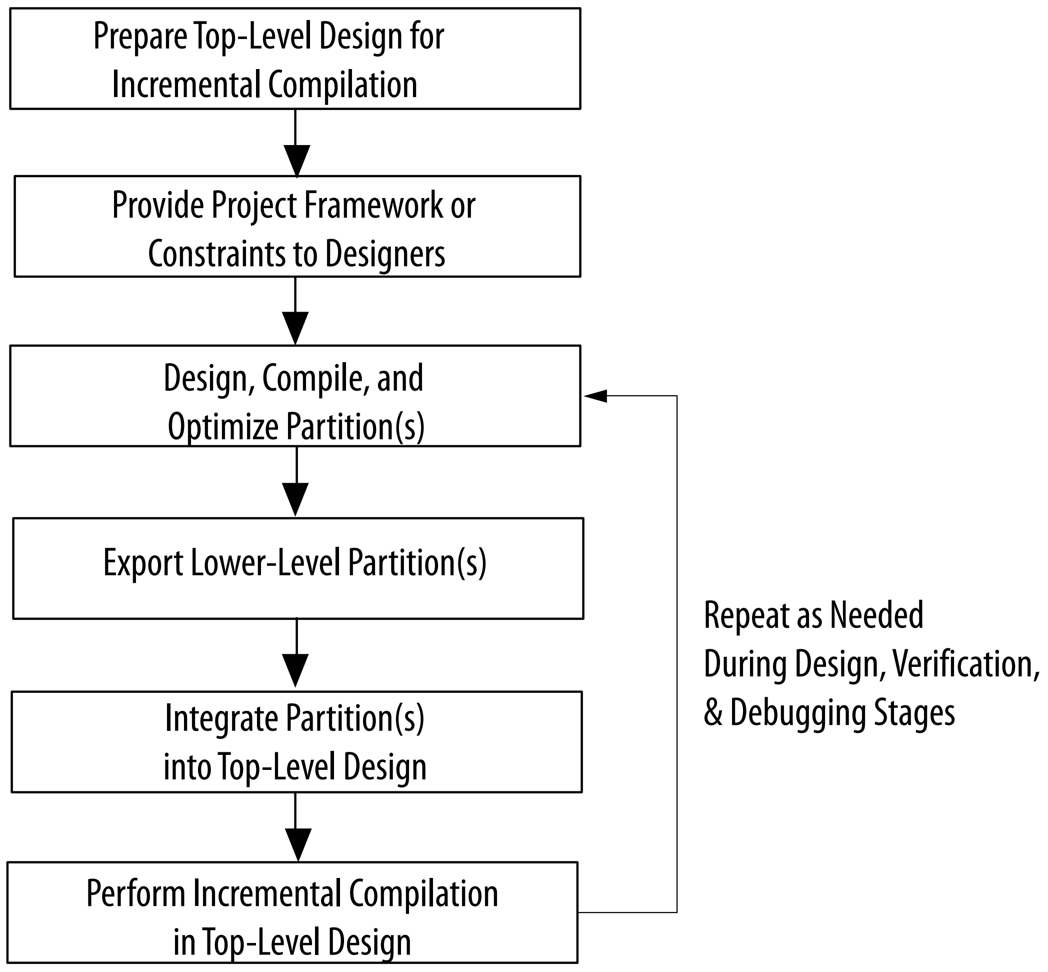
Partitions that are developed by other designers or team members in the same company or third-party IP providers can be exported as design partitions to a Quartus II Exported Partition File (**.qxp**), and then integrated into a top-level design. A **.qxp** is a binary file that contains compilation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

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

The exported compilation results of completed partitions are given to the project lead, preferably using a source control system, who is then responsible for integrating them into the top-level design to obtain a fully functional design. This type of design flow is required only if partition designers want to optimize their placement and routing independently, and pass their design to the project lead to reuse placement and routing results. Otherwise, a project lead can integrate source HDL from several designers in a single Quartus II project, and use the standard incremental compilation flow described previously.

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

Figure 3-7: Team-Based Incremental Compilation Design Flow


Prepare Top-Level Design for Incremental Compilation
Provide Project Framework or Constraints to Designers
Design, Compile, and Optimize Partition(s)
Export Lower-Level Partition(s)
Integrate Partition(s) into Top-Level Design
Perform Incremental Compilation in Top-Level Design
Repeat as Needed During Design, Verification, & Debugging Stages

Note: You cannot export or import partitions that have been merged.

Related Information

- Deciding Which Design Blocks Should Be Design Partitions on page 3-20

- Incremental Compilation Restrictions on page 3-49

Preparing the Top-Level Design

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

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

Next, create the appropriate design partition assignments and set the netlist type for each design partition that will be developed in a separate Quartus II project to **Empty**. It may be necessary to constrain the location of partitions with LogicLock region assignments if they are timing-critical and are expected to change in future compilations, or if the designer or IP provider wants to place and route their design partition independently, to avoid location conflicts.

Finally, provide the top-level project framework to the partition designers, preferably through a source control system.

Related Information

Creating a Design Floorplan With LogicLock Regions on page 3-47

Empty Partitions

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

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

When you set a design partition to **Empty**, a design file is required during Analysis and Synthesis to specify the port interface information so that it can connect the partition correctly to other logic and partitions in the design. If a partition is exported from another project, the **.qxp** contains this information. If there is no **.qxp** or design file to represent the design entity, you must create a wrapper file that defines the design block and specifies the input, output, and bidirectional ports. For example, in Verilog HDL, you should include a module declaration, and in VHDL, you should include an entity and architecture declaration.

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

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

Distributing the Top-Level Quartus II Project

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

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

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

- Ensure that a LogicLock region constrains the partition placement and uses only the resources allocated by the project lead.
- Ensure that you know which clocks should be allocated to global routing resources so that there are no resource conflicts in the top-level design.
    - Set the Global Signal assignment to **On** for the high fan-out signals that should be routed on global routing lines.
    - To avoid other signals being placed on global routing lines, turn off **Auto Global Clock and Auto Global Register Controls** under **More Settings** on the Fitter page in the **Settings** dialog box. Alternatively, you can set the Global Signal assignment to **Off** for signals that should not be placed on global routing lines.

      Placement for LABs depends on whether the inputs to the logic cells within the LAB use a global clock. You may encounter problems if signals do not use global lines in the partition, but use global routing in the top-level design.
- Use the Virtual Pin assignment to indicate pins of a partition that do not drive pins in the top-level design. This is critical when a partition has more output ports than the number of pins available in the target device. Using virtual pins also helps optimize cross-partition paths for a complete design by enabling you to provide more information about the partition ports, such as location and timing assignments.
- When partitions are compiled independently without any information about each other, you might need to provide more information about the timing paths that may be affected by other partitions in the top-level design. You can apply location assignments for each pin to indicate the port location after incorporation in the top-level design. You can also apply timing assignments to the I/O ports of the partition to perform timing budgeting.

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Generating Design Partition Scripts

If IP providers or designers on a team want to optimize their design blocks independently and do not have access to a shared project framework, the project lead must perform some or all of the following tasks to ensure successful integration of the design blocks:

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

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

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

Related Information

- **Enabling Designers on a Team to Optimize Independently** on page 3-42
- **Generating Design Partition Scripts for Project Management online help**
- **Generate Design Partition Scripts Dialog Box online help**

Exporting Partitions

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

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

When designing the partition logic to be exported into another project, you can add logic around the design block to be exported as a design partition. You can instantiate additional design components for the Quartus II project so that it matches the top-level design environment, especially in cases where you do not have access to the full top-level design project. For example, you can include a top-level PLL in the project, outside of the partition to be exported, so that you can optimize the design with information about the frequency multipliers, phase shifts, compensation delays, and any other PLL parameters. The software then captures timing and resource requirements more accurately while ensuring that the timing analysis in the partition is complete and accurate. You can export the partition for the top-level design without any auxiliary components that are instantiated outside the partition being exported.

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

Related Information

Using a Team-Based Incremental Compilation Design Flow online help

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

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

Integrating Partitions into the Top-Level Design

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

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

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

If you assigned the **.qxp** instance as a partition, you can set the netlist type in the Design Partitions Window to choose the level of results to preserve from the **.qxp**. To preserve the placement and routing results from the exported partition, set the netlist type to **Post-Fit** for the **.qxp** partition in the top-level design. If you assign the instance as a design partition, the partition boundary is preserved.

Related Information

Impact of Design Partitions on Design Optimization on page 3-22

Integrating Assignments from the .qxp

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

Design Partition Assignments Within the Exported Partition

Design partition assignments defined within a separate Quartus II project are not added to the top-level design. All logic under the exported partition in the project hierarchy is treated as single instance in the **.qxp**.

Synopsys Design Constraint Files for the Quartus II TimeQuest Timing Analyzer

Timing assignments made for the Quartus II TimeQuest analyzer in a Synopsys Design Constraint File (**.sdc**) in the lower-level partition project are not added to the top-level design. Ensure that the top-level design includes all of the timing requirements for the entire project.

Related Information

- Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation on page 14-1

Global Assignments

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

LogicLock Region Assignments

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

Integrating Encrypted IP Cores from .qxp Files

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

Advanced Importing Options

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

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

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

After you compile the entire design, if you make changes to the place-and-route results (such as movement of an imported LogicLock region), use the **Post-Fit** netlist type on subsequent compilations. To discard an imported netlist and recompile from source code, you can compile the partition with the netlist type set to **Source File** and be sure to include the relevant source code in the top-level design. The import process sets the partition's Fitter Preservation Level to the setting with the highest degree of preservation supported by the imported netlist. For example, if a post-fit netlist is imported with placement information, the Fitter Preservation Level is set to **Placement**, but you can change it to the **Netlist Only** value.

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

When you import a partition into a top-level design with the **Import Design Partition** dialog box, the software imports relevant assignments from the partition into the top-level design. If required, you can change the way some assignments are imported, as described in the following subsections.

Related Information

- Netlist Type for Design Partitions on page 3-25
- Fitter Preservation Level for Design Partitions on page 3-27

Importing LogicLock Assignments

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

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

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

Related Information

Changing Partition Placement with LogicLock Changes on page 3-48

Advanced Import Settings

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

Related Information

Advanced Import Settings Dialog Box online help

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

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

Scenario background: A designer wants to produce a design block and needs to send out their design, but to preserve their IP, they prefer to send a synthesized netlist instead of providing the HDL source code to the recipient. You can use this flow to implement a black box.

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

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

1. Provide the device family name to the recipient. If you send placement information with the synthesized netlist, also provide the exact device selection so they can set up their project to match.
2. Create a black box wrapper file that defines the port interface for the design block and provide it to the recipient for instantiating the block as an empty partition in the top-level design.
3. Create a Quartus II project for the design block, and complete the design.
4. Export the level of hierarchy into a single **.qxp**. Following a successful compilation of the project, you can generate a **.qxp** from the GUI, the command-line, or with Tcl commands, as described in the following:
    - If you are using the Quartus II GUI, use the **Export Design Partition** dialog box.
    - If you are using command-line executables, run **quartus_cdb** with the `--incremental_compilation_export` option.
    - If you are using Tcl commands, use the following command: `execute_flow -incremental_compilation_export`.
5. Select the option to include just the **Post-synthesis netlist** if you do not have to send placement information. If the recipient wants to reproduce your exact Fitter results, you can select the **Post-fitting netlist** option, and optionally enable **Export routing**.
6. If a partition contains sub-partitions, then the sub-partitions are automatically flattened and merged into the partition netlist before exporting. You can change this behavior and preserve the sub-partition hierarchy by turning off the **Flatten sub-partitions** option on the **Export Design Partition** dialog box. Optionally, you can use the `-dont_flatten` sub-option for the `export_partition` Tcl command.
7. Provide the **.qxp** to the recipient. Note that you do not have to send any of your design source code.

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

Add the **.qxp** from the IP designer as a source file in your Quartus II project to replace any empty wrapper file. If you want to use just the post-synthesis information, you can choose whether you want the file to be a partition in the top-level design. To use the post-fit information from the **.qxp**, assign the instance as a design partition and set the netlist type to **Post-Fit**.

Related Information

- Creating Design Partitions on page 3-9
- Netlist Type for Design Partitions on page 3-25

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

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

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

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

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

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

   Using a LogicLock region for the IP core allows the customer to create an empty placeholder region to reserve space for the IP in the design floorplan and ensures that there are no conflicts with the top-level design logic. Reserved space also helps ensure the IP core does not affect the timing performance of other logic in the top-level design. Additionally, with a LogicLock region, you can preserve placement either absolutely or relative to the origin of the associated region. This is important when a **.qxp** is imported for multiple partition hierarchies in the same project, because in this case, the location of at least one instance in the top-level design does not match the location used by the IP provider.
4. If required, add any logic (such as PLLs or other logic defined in the customer's top-level design) around the design hierarchy to be exported. If you do so, create a design partition for the design hierarchy that will exported as an IP core.
5. Optimize the design and close timing to meet the design specifications.
6. Export the level of hierarchy for the IP core into a single **.qxp**.
7. Provide the **.qxp** to the customer. Note that you do not have to send any of your design source code to the customer; the design netlist and placement and routing information is contained within the **.qxp**.

Related Information

- **Creating Design Partitions** on page 3-55
- **Netlist Type for Design Partitions** on page 3-25
- **Changing Partition Placement with LogicLock Changes** on page 3-48

Incorporate IP Core

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

1. Create a Quartus II project for the top-level design that targets the same device and instantiate a copy or multiple copies of the IP core. Use a black box wrapper file to define the port interface of the IP core.
2. Perform Analysis and Elaboration to identify the design hierarchy.
3. Create a design partition for each instance of the IP core with the netlist type set to **Empty**.
4. You can now continue work on your part of the design and accept the IP core from the IP provider when it is ready.
5. Include the **.qxp** from the IP provider in your project to replace the empty wrapper-file for the IP instance. Or, if you are importing multiple copies of the design block and want to import relative placement, follow these additional steps:
   a. Use the **Import** command to select each appropriate partition hierarchy. You can import a **.qxp** from the GUI, the command-line, or with Tcl commands:
      - If you are using the Quartus II GUI, use the **Import Design Partition** command.
      - If you are using command-line executables, run **quartus_cdb** with the `incremental_compilation_import` option.
      - If you are using Tcl commands, use the following command:`execute_flow -incremental_compilation_import`.

   b. When you have multiple instances of the IP block, you can set the imported LogicLock regions to floating, or move them to a new location, and the software preserves the relative placement for each of the imported modules (relative to the origin of the LogicLock region). Routing information is preserved whenever possible.

      Note: The Fitter ignores relative placement assignments if the LogicLock region's location in the top-level design is not compatible with the locations exported in the **.qxp**.
6. You can control the level of results preservation with the **Netlist Type** setting.

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

Designing in a Team-Based Environment

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

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

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

1. Create a new Quartus II project to ultimately contain the full implementation of the entire design and include a "skeleton" or framework of the design that defines the hierarchy for the subdesigns implemented by separate designers. The top-level design implements the top-level entity in the design

and instantiates wrapper files that represent each subdesign by defining only the port interfaces, but not the implementation.

2. Make project-wide settings. Select the device, make global assignments such as device I/O ports, define the top-level timing constraints, and make any global signal allocation constraints to specify which signals can use global routing resources.
3. Make design partition assignments for each subdesign and set the netlist type for each design partition to be imported to **Empty** in the Design Partitions window.
4. Create LogicLock regions to create a design floorplan for each of the partitions that will be developed separately. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications.
5. Provide the top-level project framework to partition designers using one of the following procedures:

    - Allow access to the full project for all designers through a source control system. Each designer can check out the projects files as read-only and work on their blocks independently. This design flow provides each designer with the most information about the full design, which helps avoid resource conflicts and makes design integration easy.
    - Provide a copy of the top-level Quartus II project framework for each designer. You can use the **Copy Project** command on the Project menu or create a project archive.

Exporting Your Partition

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

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

Integrating Your Partitions

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

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

Enabling Designers on a Team to Optimize Independently

Scenario background: A project consists of several lower-level design blocks that are developed separately by different designers who do not have access to a shared top-level project framework. This scenario is similar to creating precompiled design blocks for resuse, but assumes that there are several design blocks being developed independently (instead of just one IP block), and the project lead can provide some information about the design to the individual designers. If the designers have shared access to the top-level design, use the instructions for designing in a team-based environment.

This scenario assumes that there are several design blocks being developed independently (instead of just one IP block), and the project lead can provide some information about the design to the individual designers.

This scenario describes how to use incremental compilation in a team-based design environment where designers or IP developers want to fully optimize the placement and routing of their design independently in a separate Quartus II project before sending the design to the project lead. This design flow requires more planning and careful resource allocation because design blocks are developed independently.

Related Information

- **Creating Precompiled Design Blocks (or Hard-Wired Macros) for Reuse** on page 3-40
- **Designing in a Team-Based Environment** on page 3-41

Preparing Your Top-level Design

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

1. Create a new Quartus II project to ultimately contain the full implementation of the entire design and include a "skeleton" or framework of the design that defines the hierarchy for the subdesigns implemented by separate designers. The top-level design implements the top-level entity in the design and instantiates wrapper files that represent each subdesign by defining only the port interfaces but not the implementation.
2. Make project-wide settings. Select the device, make global assignments such as device I/O ports, define the top-level timing constraints, and make any global signal constraints to specify which signals can use global routing resources.
3. Make design partition assignments for each subdesign and set the netlist type for each design partition to be imported to **Empty** in the Design Partitions window.
4. Create LogicLock regions. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications.
5. Provide the constraints from the top-level design to partition designers using one of the following procedures.

    - Use design partition scripts to pass constraints and generate separate Quartus II projects. On the Project menu, use the **Generate Design Partition Scripts** command, or run the script generator from a Tcl or command prompt. Make changes to the default script options as required for your project. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. If partitions have not already been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles. Provide each partition designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.
    - Use documentation or manually-created scripts to pass all constraints and assignments to each partition designer.

Exporting Your Design

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

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

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

Exporting Without Makefiles

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

1. If you are using design partition scripts, source the Tcl script provided by the Project Lead to create a project with the required settings:
    - To source the Tcl script in the Quartus II software, on the Tools menu, click **Utility Windows** to open the Tcl console. Navigate to the script's directory, and type the following command: `source <filename>`.
    - To source the Tcl script at the system command prompt, type the following command: `quartus_cdb -t <filename>.tcl`
2. If you are not using design partition scripts, create a new Quartus II project for the subdesign, and then apply the following settings and constraints to ensure successful integration:
    - Make LogicLock region assignments and global assignments (including clock settings) as specified by the project lead.
    - Make Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level design.
    - Make floorplan location assignments to the Virtual Pins so they are placed in their corresponding regions as determined by the top-level design. This provides the Fitter with more information about the timing constraints between modules. Alternatively, you can apply timing I/O constraints to the paths that connect to virtual pins.
3. Proceed to compile and optimize the design as needed.
4. When you have achieved the desired compilation results, on the Project menu, click **Export Design Partition**.
5. In the **Export Design Partition** dialog box, choose the netlist(s) to export. You can export a Post-synthesis netlist instead if placement or performance preservation is not required, to provide the most flexibility for the Fitter in the top-level design. Select **Post-fit** to preserve the placement and performance of the lower-level design block, and turn on Export routing to include the routing information, if required. One **.qxp** can include both post-synthesis and post-fitting netlists.
6. Provide the **.qxp** to the project lead.

Importing Your Design

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

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

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

Importing Without Makefiles

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

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

Resolving Assignment Conflicts During Integration

When integrating lower-level design blocks, the project lead may notice some assignment conflicts. This can occur, for example, if the lower-level design block designers changed their LogicLock regions to account for additional logic or placement constraints, or if the designers applied I/O port timing constraints that differ from constraints added to the top-level design by the project lead. The project lead can address these conflicts by explicitly importing the partitions into the top-level design, and using options in the **Advanced Import Settings** dialog box. After the project lead obtains the **.qxp** for each lower-level design block from the other designers, use the **Import Design Partition** command on the Project menu and specify the partition in the top-level design that is represented by the lower-level design block **.qxp**. Repeat this import process for each partition in the design. After you have imported each partition once, you can select all the design partitions and use the **Reimport using latest import files at previous locations** option to import all the files from their previous locations at one time. To address assignment conflicts, the project lead can take one or both of the following actions:

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

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

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

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

Importing a Partition to be Instantiated Multiple Times

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

If you import multiple instances of a lower-level design block into the top-level design, the imported LogicLock regions are automatically set to Floating status.

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

Performing Design Iterations With Lower-Level Partitions

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

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

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

Providing the Complete Top-Level Project Framework

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

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

Providing Information About the Top-Level Framework

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

1. In the top-level design, on the Project menu, click **Generate Design Partition Scripts**, or launch the script generator from Tcl or the command line.
2. If lower-level projects have already been created for each partition, you can turn off the **Create lower-level project if one does not exist** option.
3. Make additional changes to the default script options, as necessary. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera also recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition.
4. The Quartus II software generates Tcl scripts for all partitions, but in this scenario, you would focus on the partitions that make up the cross-partition critical paths. The following assignments are important in the script:

    - Virtual pin assignments for module pins not connected to device I/O ports in the top-level design.

- Location constraints for the virtual pins that reflect the initial top-level placement of the pin's source or destination. These help make the lower-level placement "aware" of its surroundings in the top-level design, leading to a greater chance of timing closure during integration at the top level.
- `INPUT_MAX_DELAY` and `OUTPUT_MAX_DELAY` timing constraints on the paths to and from the I/O pins of the partition. These constrain the pins to optimize the timing paths to and from the pins.

5. The partition designers source the file provided by the project lead.

- To source the Tcl script from the Quartus II GUI, on the Tools menu, click **Utility Windows** and open the Tcl console. Navigate to the script's directory, and type the following command:

```
source <filename>
```

- To source the Tcl script at the system command prompt, type the following command:

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

6. The partition designers recompile their designs with the new project information or assignments and optimize as needed. When the results are satisfactory and the timing requirements are met, export the updated partition as a **.qxp**.

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

You can now analyze the design to determine whether the timing requirements have been achieved. Because the partitions were compiled with more information about connectivity at the top level, it is more likely that the inter-partition paths have improved placement which helps to meet the timing requirements.

Creating a Design Floorplan With LogicLock Regions

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

- To avoid resource conflicts between partitions, predominantly when partitions are imported from another Quartus II project
- To ensure a good quality of results when recompiling individual timing-critical partitions

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

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

The simplest way to create a floorplan for a partitioned design is to create one LogicLock region per partition (including the top-level partition). If you have a compilation result for a partitioned design with no LogicLock regions, you can use the Chip Planner with the Design Partition Planner to view the partition placement in the device floorplan. You can draw regions in the floorplan that match the general location and size of the logic in each partition. Or, initially, you can set each region with the default settings of **Auto** size and **Floating** location to allow the Quartus II software to determine the preliminary size and location for the regions. Then, after compilation, use the Fitter-determined size and origin location as a starting point for your design floorplan. Check the quality of results obtained for your floorplan location assignments and make changes to the regions as needed. Alternatively, you can perform synthesis, and then set the regions to the required size based on resource estimates. In this case, use your knowledge of the connections between partitions to place the regions in the floorplan.

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

You can use the Incremental Compilation Advisor to check that your LogicLock regions meet Altera's guidelines.

Related Information

Incremental Compilation Advisor on page 3-24

Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation on page 14-1

Creating and Manipulating LogicLock Regions

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

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

Related Information

Creating and Manipulating LogicLock Regions online help

Changing Partition Placement with LogicLock Changes

When a partition is assigned to a LogicLock region as part of a design floorplan, you can modify the placement of a post-fit partition by moving the LogicLock region. Most assignment changes do not initiate a recompilation of a partition if the netlist type specifies that Fitter results should be preserved. For example, changing a pin assignment does not initiate a recompilation; therefore, the design does not use the new pin assignment unless you change the netlist type to **Post Synthesis** or **Source File**.

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

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

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

Related Information

What Changes Initiate the Automatic Resynthesis of a Partition? on page 3-29

Incremental Compilation Restrictions

When Timing Performance May Not Be Preserved Exactly

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

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

When Placement and Routing May Not Be Preserved Exactly

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

Using Incremental Compilation With Quartus II Archive Files

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

If you want to re-use post-synthesis or post-fitting results, include the database files in the **Archive Project** dialog box so compilation results are preserved. Click **Advanced**, and choose a file set that includes the compilation database, or turn on **Incremental compilation database files** to create a Custom file set.

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

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

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

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

Formal Verification Support

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

SignalProbe Pins and Engineering Change Orders

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

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

For partitions developed independently in separate Quartus II projects, the exported netlist includes all currently saved ECO changes and SignalProbe signals. If you make any ECO or SignalProbe changes that affect the interface to the lower-level partition, the software issues a warning message during the export process that this netlist does not work in the top-level design without modifying the top-level HDL code to reflect the lower-level change. After integrating the **.qxp** partition into the top-level design, the ECO changes will not appear in the Change Manager.

Related Information

- **Quick Design Debugging Using SignalProbe documentation**
- **Engineering Change Management with the Chip Planner documentation**

SignalTap II Logic Analyzer in Exported Partitions

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

When incremental compilation is turned on, debugging logic is added to your design incrementally and you can tap post-fitting nodes and modify triggers and configuration without recompiling the full design. Use the appropriate filter in the Node Finder to find your node names. Use **SignalTap II: post-fitting** if the netlist type is Post-Fit to incrementally tap node names in the post-fit netlist database. Use **SignalTap II: pre-synthesis** if the netlist type is **Source File** to make connections to the source file (pre-synthesis) node names when you synthesize the partition from the source code.

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

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

You can also instantiate the SignalTap II IP core directly in your lower-level design (instead of using an **.stp** file) and export the entire design to the top level to include the logic analyzer in the top-level design.

Related Information

Design Debugging Using the SignalTap II Embedded Logic Analyzer documentation

External Logic Analyzer Interface in Exported Partitions

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

Related Information

In-System Debugging Using External Logic Analyzers documentation

Assignments Made in HDL Source Code in Exported Partitions

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

Design Partition Script Limitations

Related Information

Generating Design Partition Scripts on page 3-35

Warnings About Extra Clocks Due to Design Partition Scripts

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

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

After you have compiled a design using TimeQuest constraints, and the timing assignments option is turned on in the scripts, a separate Tcl script is generated to create an **.sdc** file for each lower-level project.

This script includes only clock constraints and minimum and maximum delay settings for the TimeQuest Timing Analyzer.

Note: PLL settings and timing exceptions are not passed to lower-level designs in the scripts.

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Wildcard Support in Design Partition Scripts

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

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

Derived Clocks and PLLs in Design Partition Scripts

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

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

If the lower-level design uses the top-level project framework from the project lead, the design will have all the required information about the clock and PLL settings. Otherwise, if you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication or phase shift factors in the PLL. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained or constrained with the incorrect frequency. Alternatively, you can manually duplicate the top-level derived clock logic or PLL in the lower-level design file to ensure that you have the correct multiplication or phase-shift factors, compensation delays and other PLL parameters for complete and accurate timing analysis. Create a design partition for the rest of the lower-level design logic for export to the top level. When the lower-level design is complete, export only the partition that contains the relevant logic.

Pin Assignments for GXB and LVDS Blocks in Design Partition Scripts

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

Virtual Pin Timing Assignments in Design Partition Scripts

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

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

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

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

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

Restrictions on IP Core Partitions

The Quartus II software does not support partitions for IP core instantiations. If you use the parameter editor to customize an IP core variation, the IP core generated wrapper file instantiates the IP core. You can create a partition for the IP core generated wrapper file.

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

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

Register Packing and Partition Boundaries

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

I/O Register Packing

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

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

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

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

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

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

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

The restrictions on tri-state logic exist because the I/O atom (device primitive) is created as part of the partition that contains tri-state logic. If an I/O register and its tri-state logic are contained in the same partition, the register can always be packed with tri-state logic into the I/O atom. The same cross-partition register packing restrictions also apply to I/O atoms for input and output pins. The I/O atom must feed the I/O pin directly with exactly one signal. The path between the I/O atom and the I/O pin must include only ports of partitions that have one fan-out each.

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script or at a command-line prompt.

Tcl Scripting and Command-Line Examples

The `::quartus::incremental_compilation` Tcl package contains a set of functions for manipulating design partitions and settings related to the incremental compilation feature.

Related Information

- **::quartus::incremental_compilation online help**
- **Quartus II Software Scripting Support website**
  Scripting support information, design examples, and training

- Tcl Scripting documentation
- Command-Line Scripting documentation

Creating Design Partitions

To create a design partition to a specified hierarchy name, use the following command:

Example 3-1: Create Design Partition

```
create_partition [-h | -help] [-long_help] -contents
<hierarchy name> -partition <partition name>
```

Table 3-4: Tcl Script Command: `create_partition`

Argument	Description
`-h | -help`	Short help
`-long_help`	Long help with examples and possible return values
`-contents <hierarchy name>`	Partition contents (hierarchy assigned to Partition)
`-partition <partition name>`	Partition name

Enabling or Disabling Design Partition Assignments During Compilation

To direct the Quartus II Compiler to enable or disable design partition assignments during compilation, use the following command:

Example 3-2: Enable or Disable Partition Assignments During Compilation

```
set_global_assignment -name IGNORE_PARTITIONS <value>
```

Table 3-5: Tcl Script Command: `set_global_assignment`

Value	Description
`OFF`	The Quartus II software recognizes the design partitions assignments set in the current Quartus II project and recompiles the partition in subsequent compilations depending on their netlist status.

Value	Description
`ON`	The Quartus II software does not recognize design partitions assignments set in the current Quartus II project and performs a compilation without regard to partition boundaries or netlists.

Setting the Netlist Type

To set the partition netlist type, use the following command:

Example 3-3: Set Partition Netlist Type

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

Note: The `PARTITION_NETLIST_TYPE` command accepts the following values: `SOURCE`, `POST_SYNTH`, `POST_FIT`, and `EMPTY`.

Setting the Fitter Preservation Level for a Post-fit or Imported Netlist

To set the Fitter Preservation Level for a post-fit or imported netlist, use the following command:

Example 3-4: Set Fitter Preservation Level

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

Note: The `PARTITION_FITTER_PRESERVATION` command accepts the following values: `NETLIST_ONLY`, `PLACEMENT`, and `PLACEMENT_AND_ROUTING`.

Preserving High-Speed Optimization

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

Example 3-5: Preserve High-Speed Optimization

```
set_global_assignment -name PARTITION_PRESERVE_HIGH_SPEED_TILES_ON
```

Specifying the Software Should Use the Specified Netlist and Ignore Source File Changes

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

Example 3-6: Specify Netlist and Ignore Source File Changes

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

Reducing Opening a Project, Creating Design Partitions, andPerforming an Initial Compilation

Scenario background: You open a project called `AB_project`, set up two design partitions, entities A and B, and then perform an initial full compilation.

Example 3-7: Set Up and Compile AB_project

```
set project AB_project

load_package incremental_compilation
load_package flow
project_open $project

# Ensure that design partition assignments are not ignored
set_global_assignment -name IGNORE_PARTITIONS \ OFF

# Set up the partitions
create_partition -contents A -name "Partition_A"
create_partition -contents B -name "Partition_B"

# Set the netlist types to post-fit for subsequent
# compilations (all partitions are compiled during the
# initial compilation since there are no post-fit netlists)
set_partition -partition "Partition_A" -netlist_type POST_FIT
set_partition -partition "Partition_B" -netlist_type POST_FIT

# Run initial compilation
export_assignments
execute_flow -full_compile

project_close
```

Optimizing the Placement for a Timing-Critical Partition

Scenario background: You have run the initial compilation shown in the example script below. You would like to apply Fitter optimizations, such as physical synthesis, only to partition **A**. No changes have been made to the HDL files. To ensure the previous compilation result for partition **B** is preserved, and to ensure that Fitter optimizations are applied to the post-synthesis netlist of partition **A**, set the netlist type of **B** to **Post-Fit** (which was already done in the initial compilation, but is repeated here for safety), and the netlist type of **A** to **Post-Synthesis**, as shown in the following example:

Example 3-8: Fitter Optimization for AB_project

```
set project AB_project

load_package flow
load_package incremental_compilation
load_package project
project_open $project

# Turn on Physical Synthesis Optimization
```

```
set_high_effort_fmax_optimization_assignments

# For A, set the netlist type to post-synthesis
set_partition -partition "Partition_A" -netlist_type POST_SYNTH

# For B, set the netlist type to post-fit
set_partition -partition "Partition_B" -netlist_type POST_FIT

# Also set Top to post-fit
set_partition -partition "Top" -netlist_type POST_FIT

# Run incremental compilation
export_assignments
execute_flow -full_compile

project_close
```

Generating Design Partition Scripts

To generate design partition scripts, use the following script:

Example 3-9: Generate Partition Script

```
# load required package
load_package database_manager

# name and open the project
set project <project_path/project_name>
project_open $project

# generate the design partition scripts
generate_bottom_up_scripts <options>

#close project
project_close
```

Related Information

Generate Design Partition Scripts Dialog Box online help

Exporting a Partition

To open a project and load the `::quartus::incremental_compilation` package before you use the Tcl commands to export a partition to a **.qxp** that contains both a post-synthesis and post-fit netlist, with routing, use the following script:

Example 3-10: Export .qxp

```
# load required package
load_package incremental_compilation

# open project
project_open <project name>

# export partition to the .qxp and set preservation level
export_partition -partition <partition name>
-qxp <.qxp file name> -<options>
```

```
#close project
project_close
```

Importing a Partition into the Top-Level Design

To import a **.qxp** into a top-level design, use the following script:

Example 3-11: Import .qxp into Top-Level Design

```
# load required packages
load_package incremental_compilation
load_package project
load_package flow

# open project
project_open <project name>

#import partition
import_partition -partition <partition name> -qxp <.qxp file>
<-options>

#close project
project_close
```

Makefiles

For an example of how to use incremental compilation with a `makefile` as part of the team-based incremental compilation design flow, refer to the **read_me.txt** file that accompanies the `incr_comp` example located in the **/qdesigns/incr_comp_makefile** subdirectory.

When using a team-based incremental compilation design flow, the **Generate Design Partition Scripts** dialog box can write makefiles that automatically export lower-level design partitions and import them into the top-level design whenever design files change.

Related Information

Generate Design Partition Scripts Dialog Box online help

Document Revision History

Table 3-6: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	Removed Early Timing Estimate feature support.
2014.12.15	14.1.0	• Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings. • Updated DSE II content.
2014.08.18	14.0a10.0	Added restriction about smart compilation in Arria 10 devices.

Date	Version	Changes
June 2014	14.0.0	• Dita conversion. • Replaced MegaWizard Plug-In Manager content with IP Catalog and Parameter Editor content. • Revised functional safety section. Added export and import sections.
November 2013	13.1.0	Removed HardCopy device information. Revised information about Rapid Recompile. Added information about functional safety. Added information about flattening sub-partition hierarchies.
November 2012	12.1.0	Added Turning On Supported Cross-boundary Optimizations.
June 2012	12.0.0	Removed survey link.
November 2011	11.0.1	Template update.
May 2011	11.0.0	• Updated "Tcl Scripting and Command-Line Examples".
December 2010	10.1.0	• Changed to new document template. • Reorganized Tcl scripting section. Added description for new feature: **Ignore partitions assignments during compilation** option. • Reorganized "Incremental Compilation Summary" section.
July 2010	10.0.0	• Removed the explanation of the "bottom-up design flow" where designers work completely independently, and replaced with Altera's recommendations for team-based environments where partitions are developed in the same top-level project framework, plus an explanation of the bottom-up process for including independent partitions from third-party IP designers. • Expanded the Merge command explanation to explain how it now accommodates cross-partition boundary optimizations. • Restructured Altera recommendations for when to use a floorplan. • Added "Viewing the Contents of a Quartus II Exported Partition File (.qxp)" section. • Reorganized chapter to make design flow scenarios more visible; integrated into various sections rather than at the end of the chapter.

Date	Version	Changes
October 2009	9.1.0	• Redefined the bottom-up design flow as team-based and reorganized previous design flow examples to include steps on how to pass top-level design information to lower-level designers. • Moved SDC Constraints from Lower-Level Partitions section to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook.* • Reorganized the "Conclusion" section. • Removed HardCopy APEX and HardCopy Stratix Devices section.
March 2009	9.0.0	• Split up netlist types table • Moved "Team-Based Incremental Compilation Design Flow" into the "Including or Integrating partitions into the Top-Level Design" section. • Added new section "Including or Integrating Partitions into the Top-Level Design". • Removed "Exporting a Lower-Level Partition that Uses a JTAG Feature" restriction • Other edits throughout chapter
November 2008	8.1.0	• Added new section "Importing SDC Constraints from Lower-Level Partitions" on page 2–44 • Removed the Incremental Synthesis Only option • Removed section "OpenCore Plus Feature for MegaCore Functions in Bottom-Up Flows" • Removed section "Compilation Time with Physical Synthesis Optimizations" • Added information about using a **.qxp** as a source design file without importing • Reorganized several sections • Updated Figure 2–10

Related Information

Quartus II Handbook Archive

Design Planning for Partial Reconfiguration

2015.05.04

QII5V1 Subscribe Send Feedback

The Partial Reconfiguration (PR) feature in the Quartus II software allows you to reconfigure a portion of the FPGA dynamically, while the remainder of the device continues to operate. The Quartus II software supports the PR feature for the Altera® Stratix® V device family.

This chapter assumes a basic knowledge of Altera's FPGA design flow, incremental compilation, and LogicLock™ region features available in the Quartus II software. It also assumes knowledge of the internal FPGA resources such as logic array blocks (LABs), memory logic array blocks (MLABs), memory types (RAM and ROM), DSP blocks, clock networks.

Note: For assistance with support for partial reconfiguration with the Arria® V or Cyclone® V device families, file a service request at *mySupport* using the link below.

Related Information

- **Terminology** on page 4-1
- **An Example of a Partial Reconfiguration Design** on page 4-4
- **Partial Reconfiguration Design Flow** on page 4-8
- **Implementation Details for Partial Reconfiguration** on page 4-19
- **Partial Reconfiguration with an External Host** on page 4-26
- **Partial Reconfiguration with an Internal Host** on page 4-28
- **Partial Reconfiguration Project Management** on page 4-29
- **Programming Files for a Partial Reconfiguration Project** on page 4-31
- **Partial Reconfiguration Known Limitations** on page 4-38
- **mySupport**

Terminology

The following terms are commonly used in this chapter.

project: A Quartus II project contains the design files, settings, and constraints files required for the compilation of your design.

revision: In the Quartus II software, a revision is a set of assignments and settings for one version of your design. A Quartus II project can have several revisions, and each revision has its own set of assignments and settings. A revision helps you to organize several versions of your design into a single project.

incremental compilation: This is a feature of the Quartus II software that allows you to preserve results of previous compilations of unchanged parts of the design, while changing the implementation of the parts of your design that you have modified since your previous compilation of the project. The key benefits include timing preservation and compile time reduction by only compiling the logic that has changed.

partition: You can partition your design along logical hierarchical boundaries. Each design partition is independently synthesized and then merged into a complete netlist for further stages of compilation. With the Quartus II incremental compilation flow, you can preserve results of unchanged partitions at specific preservation levels. For example, you can set the preservation levels at post-synthesis or post-fit, for iterative compilations in which some part of the design is changed. A partition is only a logical partition of the design, and does not necessarily refer to a physical location on the device. However, you may associate a partition with a specific area of the FPGA by using a floorplan assignment.

For more information on design partitions, refer to the *Best Practices for Incremental Compilation Partitions andFloorplan Assignments* chapter in the *Quartus II Handbook*.

LogicLock region: A LogicLock region constrains the placement of logic in your design. You can associate a design partition with a LogicLock region to constrain the placement of the logic in the partition to a specific physical area of the FPGA.

For more information about LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan with the Chip Planner* chapter in the *Quartus II Handbook*.

PR project: Any Quartus II design project that uses the PR feature.

PR region: A design partition with an associated contiguous LogicLock region in a PR project. A PR project can have one or more PR regions that can be partially reconfigured independently. A PR region may also be referred to as a PR partition.

static region: The region outside of all the PR regions in a PR project that cannot be reprogrammed with partial reconfiguration (unless you reprogram the entire FPGA). This region is called the static region, or fixed region.

persona: A PR region has multiple implementations. Each implementation is called a persona. PR regions can have multiple personas. In contrast, static regions have a single implementation or persona.

PR control block: Dedicated block in the FPGA that processes the PR requests, handshake protocols, and verifies the CRC.

Related Information

Best Practices for Incremental Compilation Partitions and Floorplan Assignments on page 14-1

Analyzing and Optimizing the Design Floorplan with the Chip Planner

Determining Resources for Partial Reconfiguration

You can use partial reconfiguration to configure only the resources such as LABs, embedded memory blocks, and DSP blocks in the FPGA core fabric that are controlled by configuration RAM (CRAM).

The functions in the periphery, such as GPIOs or I/O Registers, are controlled by I/O configuration bits and therefore cannot be partially reconfigured. Clock multiplexers for `GCLK` and `QCLK` are also not partially reconfigurable because they are controlled by I/O periphery bits.

Figure 4-1: Partially Reconfigurable Resources

These are the types of resource blocks in a Stratix V device.

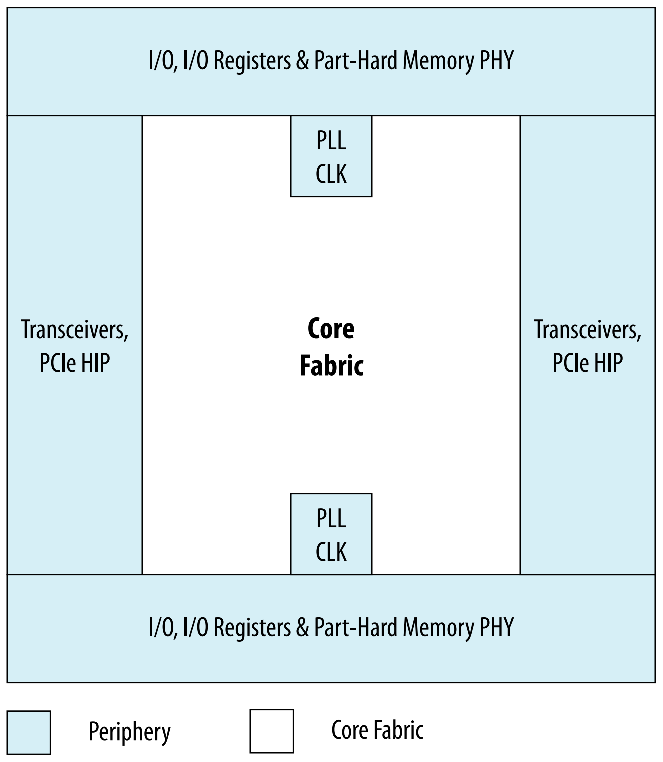


Table 4-1: Reconfiguration Modes of the FPGA Resource Block

The following table describes the reconfiguration type supported by each FPGA resource block, which are shown in the figure.

Hardware Resource Block	Reconfiguration Mode
Logic Block	Partial Reconfiguration
Digital Signal Processing	Partial Reconfiguration
Memory Block	Partial Reconfiguration
Transceivers	Dynamic Reconfiguration ALTGX_Reconfig
PLL	Dynamic Reconfiguration ALTGX_Reconfig
Core Routing	Partial Reconfiguration
Clock Networks	Clock network sources cannot be changed, but a PLL driving a clock network can be dynamically reconfigured

Hardware Resource Block	Reconfiguration Mode
I/O Blocks and Other Periphery	Not supported

The transceivers and PLLs in Altera FPGAs can be reconfigured using dynamic reconfiguration. For more information on dynamic reconfiguration, refer to the *Dynamic Reconfiguration in Stratix V Devices* chapter in the *Stratix V Handbook*.

Related Information

Dynamic Reconfiguration in Stratix V Devices

An Example of a Partial Reconfiguration Design

A PR design is divided into two parts. The static region where the design logic does not change, and one or more PR regions.

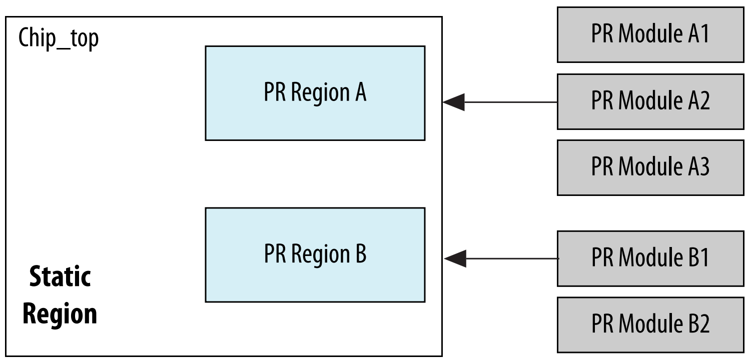
Each PR region can have different design personas, that change with partial reconfiguration.

PR Region A has three personas associated with it; A1, A2, and A3. PR Region B has two personas; B1 and B2. Each persona for the two PR regions can implement different application specific logic, and using partial reconfiguration, the persona for each PR region can be modified without interrupting the operation of the device in the static or other PR region.

When a region can access more than one persona, you must create control logic to swap between personas for a PR region.

Figure 4-2: Partial Reconfiguration Project Structure

The following figure shows the top-level of a PR design, which includes a static region and two PR regions.



Partial Reconfiguration Modes

When you implement a design on an Altera FPGA device, your design implementation is controlled by bits stored in CRAM inside the FPGA.

You can use partial reconfiguration in the SCRUB mode or the AND/OR mode. The mode you select affects your PR flow in ways detailed later in this chapter.

The CRAM bits control individual LABs, MLABs, M20K memory blocks, DSP blocks, and routing multiplexers in a design. The CRAM bits are organized into a frame structure representing vertical areas that correspond to specific locations on the FPGA. If you change a design and reconfigure the FPGA in a non-PR flow, the process reloads all the CRAM bits to a new functionality.

Configuration bitstreams used in a non-PR flow are different than those used in a PR flow. In addition to standard data and CRC check bits, configuration bitstreams for partial reconfiguration also include instructions that direct the PR control block to process the data for partial reconfiguration.

The configuration bitstream written into the CRAM is organized into configuration frames. If a LAB column passes through multiple PR regions, those regions share some programming frames.

SCRUB Mode

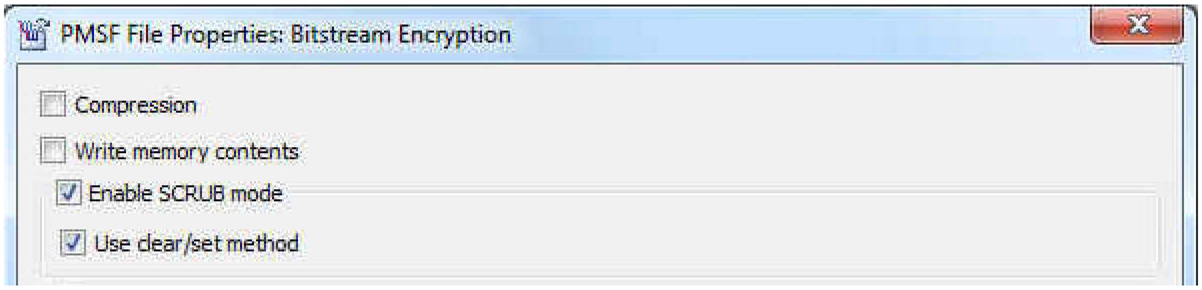
In the SCRUB mode, the unchanging CRAM bits from the static region are "scrubbed" back to their original values.

The static regions controlled by the CRAM bits from the same programming frame as the PR region continue to operate. All the CRAM bits corresponding to a PR region are overwritten with new data, regardless of what was previously contained in the region.

The SCRUB mode of partial reconfiguration involves re-writing all the bits in an entire LAB column of the CRAM, including bits controlling any part of the static region above and below the PR region boundary being reconfigured. You can choose to scrub the values of the CRAM bits to 0, and then rewrite them by turning on the **Use clear/set method** along with **Enable SCRUB mode**. The **Use clear/set method** is the more reliable option, but can increase the size of your bitstream. You can also choose to simply **Enable SCRUB mode**.

Note: You must turn on **Enable SCRUB mode** to use **Use clear/set method**.

Figure 4-3: Enable SCRUB mode and Use clear/set method


PMSF File Properties: Bitstream Encryption

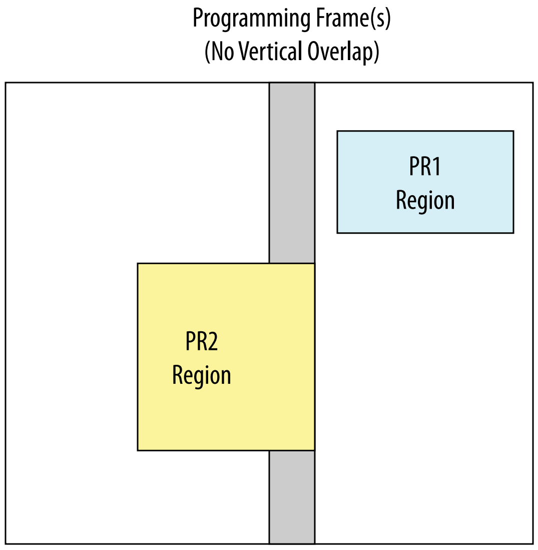
- [ ] Compression
- [ ] Write memory contents
- [x] Enable SCRUB mode
- [x] Use clear/set method

If there are more than one PR regions along a LAB column, and you are trying to reconfigure one of the PR regions, it is not not possible to correctly determine the bits associated with the PR region that is not changing. For this reason, you can not use the SCRUB mode when you have two PR regions that have a vertically overlapping column in the device. This restriction does not apply to the static bits because they never change and you can rewrite them with the same value of the configuration bit.

If you turn on **Enable SCRUB** mode and do not turn on **Use clear/set method**, then the scrub is done in a single pass, writing new values of the CRAM without clearing all the bits first. The advantage of using the SCRUB mode is that the programming file size is much smaller than the AND/OR mode.

Figure 4-4: SCRUB Mode

This is the floorplan of a FPGA using SCRUB mode, with two PR regions, whose columns do not overlap.



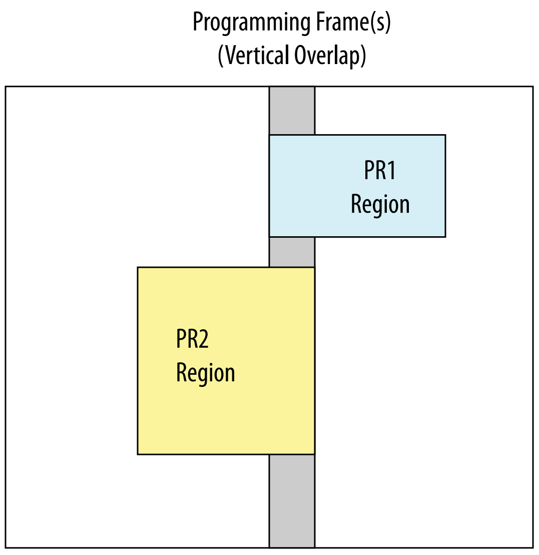
AND/OR Mode

The AND/OR mode refers to how the bits are rewritten. Partial reconfiguration with AND/OR uses a two-pass method.

Simplistically, this can be compared to bits being ANDed with a MASK, and ORed with new values, allowing multiple PR regions to vertically overlap a single column. In the first pass, all the bits in the CRAM frame for a column passing through a PR region are ANDed with 0's while those outside the PR region are ANDed with 1's. After the first pass, all the CRAM bits corresponding to the PR region are reset without modifying the static region. In the second pass for each CRAM frame, new data is ORed with the current value of 0 inside the PR region, and in the static region, the bits are ORed with 0's so they remain unchanged. The programming file size of a PR region using the AND/OR mode could be twice the programming file size of the same PR region using SCRUB mode.

Figure 4-5: AND/OR Mode

This is the floorplan of a FPGA using AND/OR mode, with two PR regions, with columns that overlap.



Note: If you have overlapping PR regions in your design, you must use AND/OR mode to program all PR regions, including PR regions with no overlap. The Quartus II software will not permit the use of SCRUB mode when there are overlapping regions. If none of your regions overlap, you can use AND/OR, SCRUB, or a mixture of both.

Programming File Sizes for a Partial Reconfiguration Project

The programming file size for a partial reconfiguration is proportional to the area of the PR region.

A partial reconfiguration programming bitstream for AND/OR mode makes two passes on the PR region; the first pass clears all relevant bits, and the second pass sets the necessary bits. Due to this two-pass sequence, the size of a partial bitstream can be larger than a full FPGA programming bitstream depending on the size of the PR region.

When using the AND/OR mode for partial reconfiguration, the formula which describes the approximate file size within ten percent is:

```
PR bitstream size = ((Size of region in the horizontal direction) /(full horizontal
dimension of the part)) * 2 * (size of full bitstream)
```

The way the Fitter reserves routing for partial reconfiguration increases the effective size for small PR regions from a bitstream perspective. PR bitstream sizes in designs with a single small PR region will not match the file size computed by this equation.

Note: The PR bitstream size is approximately half of the size computed above when using single-pass SCRUB mode. When you use the SCRUB mode with **Use clear/set method** turned on, the bitstream size is comparable to the size calculated for the AND/OR mode.

You can control expansion of the routing regions by adding the following two assignments to your Quartus II Settings file (**.qsf**):

```
set_global_assignment -name LL_ROUTING_REGION Expanded -section_id <region name>
set_global_assignment -name LL_ROUTING_REGION_EXPANSION_SIZE 0 -section_id <region
name>
```

Adding these to your **.qsf** disables expansion and minimizes the bitstream size.

Partial Reconfiguration Design Flow

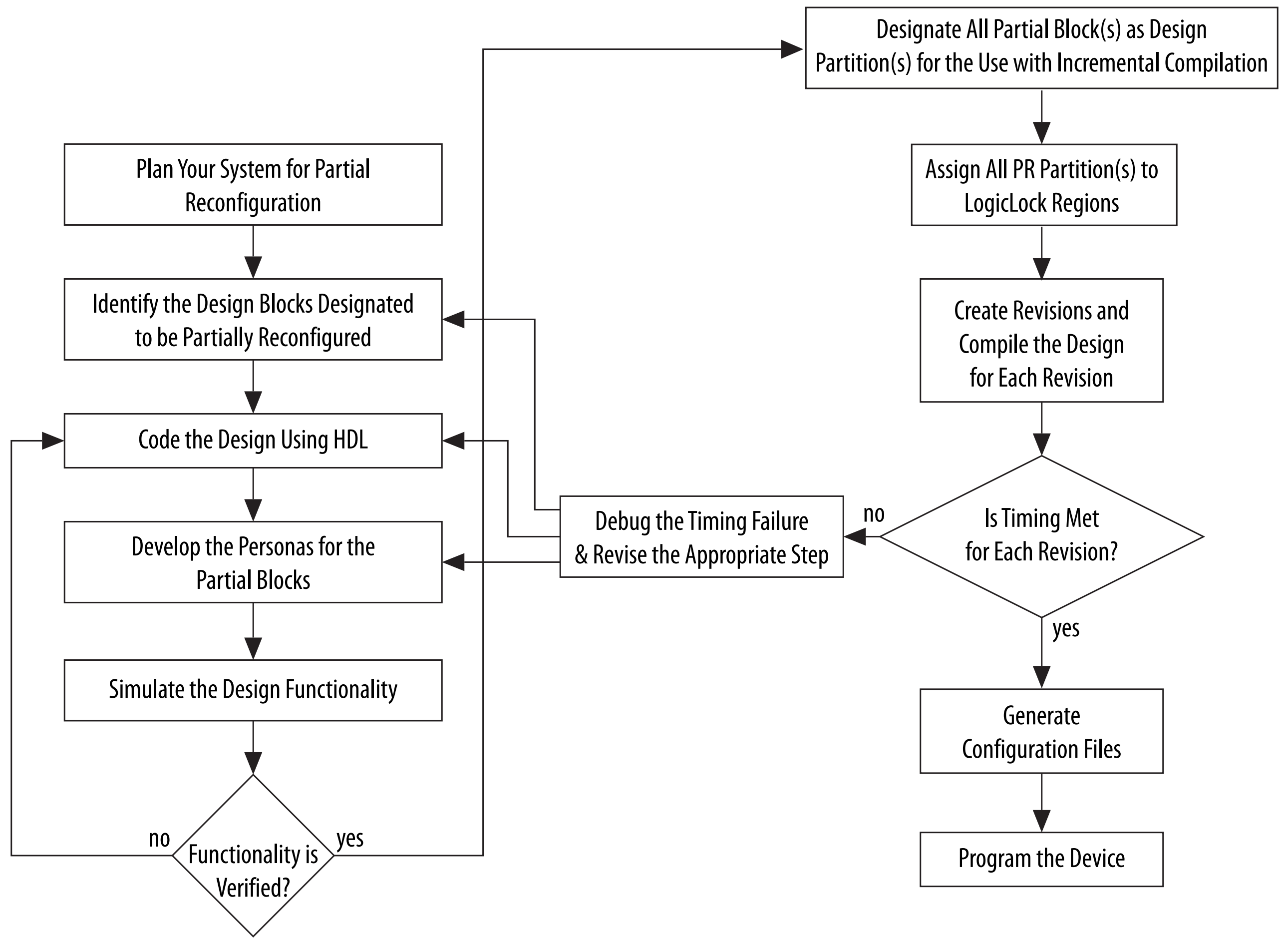
Partial reconfiguration is based on the revision feature in the Quartus II software. Your initial design is the base revision, where you define the boundaries of the static region and reconfigurable regions on the FPGA. From the base revision, you create multiple revisions, which contain the static region and describe the differences in the reconfigurable regions.

Two types of revisions are specific to partial reconfiguration: reconfigurable and aggregate. Both import the persona for the static region from the base revision. A reconfigurable revision generates personas for PR regions. An aggregate revision is used to combine personas from multiple reconfigurable revisions to create a complete design suitable for timing analysis.

The design flow for partial reconfiguration also utilizes the Quartus II incremental compilation flow. To take advantage of incremental compilation for partial reconfiguration, you must organize your design into logical and physical partitions for synthesis and fitting. For the PR flow, these partitions are treated as PR regions that must also have associated LogicLock assignments.

Revisions make use of personas, which are subsidiary archives describing the characteristics of both static and reconfigurable regions, that contain unique logic which implements a specific set of functions to reconfigure a PR region of the FPGA. Partial reconfiguration uses personas to pass this logic from one revision to another.

Figure 4-6: Partial Reconfiguration Design Flow


Plan Your System for Partial Reconfiguration
Identify the Design Blocks Designated to be Partially Reconfigured
Code the Design Using HDL
Develop the Personas for the Partial Blocks
Simulate the Design Functionality
Functionality is Verified?
no
yes
Designate All Partial Block(s) as Design Partition(s) for the Use with Incremental Compilation
Assign All PR Partition(s) to LogicLock Regions
Create Revisions and Compile the Design for Each Revision
Is Timing Met for Each Revision?
no
Debug the Timing Failure & Revise the Appropriate Step
yes
Generate Configuration Files
Program the Device

The PR design flow requires more initial planning than a standard design flow. Planning requires setting up the design logic for partitioning, and determining placement assignments to create a floorplan. Well-planned partitions can help improve design area utilization and performance, and make timing closure easier. You should also decide whether your system requires partial reconfiguration to originate from the FPGA pins or internally, and which mode you are using; the AND/OR mode or the SCRUB mode, because this influences some of the planning steps described in this section.

You must structure your source code or design hierarchy to ensure that logic is grouped correctly for optimization. Implementing the correct logic grouping early in the design cycle is more efficient than restructuring the code later. The PR flow requires you to be more rigorous about following good design practices. The guidelines for creating partitions for incremental compilation also include creating partitions for partial reconfiguration.

Use the following best practice guidelines for designing in the PR flow, which are described in detail in this section:

- Determining resources for partial reconfiguration
- Partitioning the design for partial reconfiguration
- Creating incremental compilation partitions for partial reconfiguration
- Instantiating the PR controller in the design
- Creating wrapper logic for PR regions
- Creating freeze logic for PR regions
- Planning clocks and other global signals for the PR design
- Creating floorplan assignments for the PR design

Design Partitions for Partial Reconfiguration

You must create design partitions for each PR region that you want to partially reconfigure. Optionally, you can also create partitions for the static parts of the design for timing preservation and/or for reducing compilation time.

There is no limit on the number of independent partitions or PR regions you can create in your design. You can designate any partition as a PR partition by enabling that feature in the LogicLock Regions window in the Quartus II software.

Partial reconfiguration regions do not support the following IP blocks that require a connection to the JTAG controller:

- In-System Memory Content EditorI
- In-System Signals & Probes
- Virtual JTAG
- Nios II with debug module
- SignalTap II tap or trigger sources

Note: PR partitions can contain only core resources, they cannot contain I/O or periphery elements.

Incremental Compilation Partitions for Partial Reconfiguration

Use the following best practices guidelines when creating partitions for PR regions in your design:

- Register all partition boundaries; register all inputs and outputs of each partition when possible. This practice prevents any delay penalties on signals that cross partition boundaries and keeps each register-to-register timing path within one partition for optimization.
- Minimize the number of paths that cross partition boundaries.
- Minimize the timing-critical paths passing in or out of PR regions. If there are timing-critical paths that cross PR region boundaries, rework the PR regions to avoid these paths.
- The Quartus II software can optimize some types of paths between design partitions for non-PR designs. However, for PR designs, such inter-partition paths are strictly not optimized.

For more information about incremental compilation, refer to the following chapter in the *Quartus II Handbook*.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1

Partial Reconfiguration Controller Instantiation in the Design

This section deal ing with instantiation of the partial reconfiguration controller applies only if your design requires partial reconfiguration in external host mode. Please refer to the Partial Reconfiguration with an External Host topic for more details. If you perform PR in internal host mode, you do not have to instantiate the PR control block and the CRC block, since they are are instantiated for you by the PR IP core.

You must instantiate the Stratix V PR control block and the Stratix V CRC block in your design in order to use the PR feature in Stratix V devices. You may find that adding the PR control block and CRC block at the top level of the design offers the most convenience.

For example, in a design named Core_Top, all the logic is contained under the Core_Top module hierarchy. Create a wrapper (Chip_Top) at the top-level of the hierarchy that instantiates this Core_Top module, the Stratix V PR control block, and the Stratix V CRC check modules.

If you are performing partial reconfiguration from pins, then the required pins should be on the I/O list for the top-level (Chip_Top) of the project, as shown in the code in the following examples. If you are performing partial reconfiguration from within the core, you may choose another configuration scheme, such as Active Serial, to transmit the reconfiguration data into the core, and then assemble it to 16-bit wide data inside the FPGA within your logic. In such cases, the PR pins are not part of the FPGA I/O.

Note: Verilog HDL does not require a component declaration. You can instantiate the PR control block as shown in the following example.

Related Information

Partial Reconfiguration with an External Host on page 4-26
For more information about using partial reconfiguration with and external host.

Component Declaration of the PR Control Block and CRC Block in VHDL

This code sample has the component declaration in VHDL, showing the ports of the Stratix V PR control block and the Stratix V CRC block. In the following example, the PR function is performed from within the core (code located in Core_Top) and you must add additional ports to Core_Top to connect to both components.

```
-- The Stratix V control block interface

component stratixv_prblock is
            port(
      corectl: in STD_LOGIC ;
      prrequest: in STD_LOGIC ;
      data: in STD_LOGIC_VECTOR(15 downto 0);
      error: out STD_LOGIC ;
      ready: out STD_LOGIC ;
      done: out STD_LOGIC
            ) ;
end component ;

-- The Stratix V CRC block for diagnosing CRC errors

component stratixv_crcblock is
port(
            shiftnld: in STD_LOGIC ;
   clk: in STD_LOGIC ;
   crcerror: out STD_LOGIC
) ;
end component ;
```

The following rules apply when connecting the PR control block to the rest of your design:

- The `corectl` signal must be set to '1' (when using partial reconfiguration from core) or to '0' (when using partial reconfiguration from pins).
- The `corectl` signal has to match the **Enable PR pins** option setting in the Device and Pin Options dialog box on the Setting page; if you have turned on **Enable PR pins**, then the `corectl` signal on the PR control block instantiation must be toggled to '0'.
- When performing partial reconfiguration from pins the Quartus II software automatically assigns the PR unassigned pins. If you so choose, you can make pin assignments to all the dedicated PR pins in **Pin Planner** or **Assignment Editor**.
- When performing partial reconfiguration from core, you can connect the `prblock` signals to either core logic or I/O pins, excluding the dedicated programming pin such as `DCLK`.

Instantiating the PR Control Block and CRC Block in VHDL

This code example instantiates a PR control block in VHDL, inside your top-level project, Chip_Top:

```
module Chip_Top (
//User I/O signals (excluding PR related signals)
..
..
//PR interface & configuration signals
    pr_request,
    pr_ready,
    pr_done,
    crc_error,
    dclk,
    pr_data,
    init_done
);
//user I/O signal declaration
..
..
//PR interface and configuration signals declaration
input pr_request;
output pr_ready;
output pr_done;
output crc_error;
input dclk;
input [15:0] pr_data;
output init_done

// Following shows the connectivity within the Chip_Top module
Core_Top : Core_Top
port_map (
    ..
    ..
);

m_pr : stratixv_prblock
port map(
clk => dclk,
corectl    =>  '0', //1 - when using PR from inside
                    //0 - for PR from pins; You must also enable
                    // the appropriate option in Quartus II settings
prrequest    =>  pr_request,
data        =>  pr_data,
error        =>  pr_error,
ready        =>  pr_ready,
done        =>  pr_done
);
m_crc : stratixv_crcblock
port map(
    shiftnld=>  '1', //If you want to read the EMR register when
```

```
        clk=>  dummy_clk,     //error occurrs, refer to AN539 for the
                                 //connectivity forthis signal. If you only want
                                 //to detect CRC errors, but plan to take no
                                 //further action, you can tie the shiftnld
                                 //signal to logical high.
    crcerror    =>  crc_error
    );
```

For more information on port connectivity for reading the Error Message Register (EMR), refer to the following application note.

Related Information

AN539: Test Methodology of Error Detection and Recovery using CRC in Altera FPGA Devices

Instantiating the PR Control Block and CRC Block in Verilog HDL

The following example instantiates a PR control block in Verlilog HDL, inside your top-level project, Chip_Top:

```
 module Chip_Top (

//User I/O signals (excluding PR related signals)
..
..
//PR interface & configuration signals
 pr_request,
 pr_ready,
 pr_done,
 crc_error,
 dclk,
 pr_data,
 init_done
);

//user I/O signal declaration
..
..
//PR interface and configuration signals declaration
input pr_request;
output pr_ready;
output pr_done;
output crc_error;
input dclk;
input [15:0] pr_data;
output init_done

stratixv_prblock stratixv_prblock_inst
(
.clk(dclk),
.corectl(1'b0),
.prrequest(pr_request),
.data(pr_data),
.error(pr_error),
.ready(pr_ready),
.done(pr_done)
);

stratixv_crcblock stratixv_crcblock_inst
(
.clk(clk),
.shiftnld(1'b1),
.crcerror(crc_error)
);
endmodule
```

For more information on port connectivity for reading the Error Message Register (EMR), refer to the following application note.

Related Information

AN539: Test Methodology of Error Detection and Recovery using CRC in Altera FPGA Devices

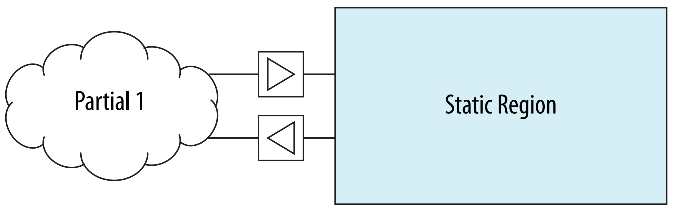
Wrapper Logic for PR Regions

Each persona of a PR region must implement the same input and output boundary ports. These ports act as the boundary between static and reconfigurable logic.

Implementing the same boundary ports ensures that all ports of a PR region remain stationary regardless of the underlying persona, so that the routing from the static logic does not change with different PR persona implementations.

Figure 4-7: Wire-LUTs at PR Region Boundary

The Quartus II software automatically instantiates a wire-LUT for each port of the PR region to lock down the same location for all instances of the PR persona.



If one persona of your PR region has a different number of ports than others, then you must create a wrapper so that the static region always communicates with this wrapper. In this wrapper, you can create dummy ports to ensure that all of the PR personas of a PR region have the same connection to the static region.

The sample code below each create two personas; `persona_1` and `persona_2` are different functions of one PR region. Note that one persona has a few dummy ports. The first example creates partial reconfiguration wrapper logic in Verilog HDL:

```
// Partial Reconfiguration Wrapper in Verilog HDL
module persona_1
(
       input reset,
       input [2:0] a,
       input [2:0] b,
       input [2:0] c,
       output [3:0] p,
       output [7:0] q
);
       reg [3:0] p, q;
       always@(a or b) begin
       p = a + b ;
end

always@(a or b or c or p)begin
       q = (p*a - b*c )
end
```

```
endmodule

module persona_2
(
       input reset,
    input [2:0] a,
       input [2:0] b,
       input [2:0] c, //never used in this persona
       output [3:0] p,
       output [7:0] q  //never assigned in this persona
);
       reg [3:0] p, q;
       always@(a or b) begin
       p = a * b;
// note q is not assigned value in this persona
end
endmodule
```

The following example creates partial reconfiguration wrapper logic in VHDL.

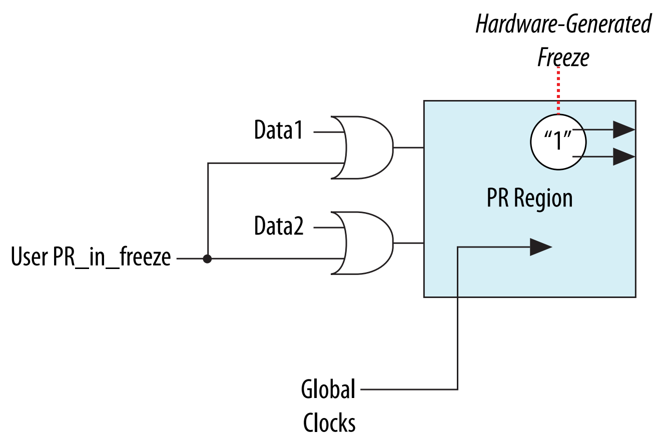
```
-- Partial Reconfiguration Wrapper in VHDL
entity persona_1 is
       port( a:in STD_LOGIC_VECTOR (2 downto 0);
                      b:in STD_LOGIC_VECTOR (2 downto 0);
                               c:in STD_LOGIC_VECTOR (2 downto 0);
                               p: out STD_LOGIC_VECTOR (3 downto 0);
                               q: out STD_LOGIC_VECTOR (7 downto 0));
end persona_1;

architecture synth of persona_1 is
    begin
                process(a,b)
                                begin
                             p <= a + b;
       end process;


       process (a, b, c, p)
        begin
        q <= (p*a - b*c);
    end process;
end synth;

entity persona_2 is
   port( a:in STD_LOGIC_VECTOR (2 downto 0);
      b:in STD_LOGIC_VECTOR (2 downto 0);
            c:in STD_LOGIC_VECTOR (2 downto 0); --never used in this persona
            p:out STD_LOGIC_VECTOR (3 downto 0);
            q:out STD_LOGIC_VECTOR (7 downto 0)); --never used in this persona

end persona_2;

architecture synth of persona_2 is
    begin
       process(a, b)
             begin
             p <= a *b; --note q is not assigned a value in this persona
       end process;
end synth;
```

Freeze Logic for PR Regions

When you use partial reconfiguration, you must freeze all non-global inputs of a PR region except global clocks. Locally routed signals are not considered global signals, and must also be frozen during partial reconfiguration. Freezing refers to driving a '1' on those PR region inputs. When you start a partial reconfiguration process, the chip is in user mode, with the device still running.

Freezing all non-global inputs for the PR region ensures there is no contention between current values that may result in unexpected behavior of the design after partial reconfiguration is complete. Global signals going into the PR region should not be frozen to high. The Quartus II software freezes the outputs from the PR region; therefore the logic outside of the PR region is not affected.

Figure 4-8: Freezing at PR Region Boundary



During partial reconfiguration, the static region logic should not depend on the outputs from PR regions to be at a specific logic level for the continued operation of the static region.

The easiest way to control the inputs to PR regions is by creating a wrapper around the PR region in RTL. In addition to freezing all inputs high, you can also drive the outputs from the PR block to a specific value, if required by your design. For example, if the output drives a signal that is active high, then your wrapper could freeze the output to GND.

The following example implements a freeze wrapper in Verilog HDL, on a module named `pr_module`.

```
module freeze_wrapper
(
 input reset,
 input freeze, //PR process active, generated by user logic
 input clk1, //global clock signal
 input clk2, // non-global clock signal
 input [3:0] control_mode,
 input [3:0] framer_ctl,
 output [15:0] data_out
);
wire [3:0]control_mode_wr, framer_ctl_wr;
wire clk2_to_wr;
//instantiate pr_module
pr_module pr_module
(
 .reset (reset), //input
```

```
  .clk1 (clk1), //input, global clock
  .clk2 (clk2_to_wr), // input, non-global clock
  .control_mode (control_mode_wr), //input
  .framer_ctl (framer_ctl_wr), //input
  .pr_module_out (data_out)// collection of outputs from pr_module
 );

 // Freeze all inputs

 assign control_mode_wr = freeze ? 4'hF: control_mode;
 assign framer_ctl_wr = freeze ? 4'hF: framer_ctl;
 assign clk2_to_wr = freeze ? 1'b1 : clk2;

 endmodule
```

The following example implements a freeze wrapper in VHDL, on a module named `pr_module`.

```
entity freeze_wrapper is
port( reset:in STD_LOGIC;
      freeze:in STD_LOGIC;
      clk1: in STD_LOGIC; --global signal
      clk2: in STD_LOGIC; --non-global signal
      control_mode: in STD_LOGIC_VECTOR (3 downto 0);
      framer_ctl: in STD_LOGIC_VECTOR (3 downto 0);
      data_out: out STD_LOGIC_VECTOR (15 downto 0));
end freeze_wrapper;

architecture behv of freeze_wrapper is

  component pr_module
  port(reset:in STD_LOGIC;
    clk1:in STD_LOGIC;
    clk2:in STD_LOGIC;
    control_mode:in STD_LOGIC_VECTOR (3 downto 0);
    framer_ctl:in STD_LOGIC_VECTOR (3 downto 0);
    pr_module_out:out STD_LOGIC_VECTOR (15 downto 0));
  end component

 signal control_mode_wr: in STD_LOGIC_VECTOR (3 downto 0);
 signal framer_ctl_wr : in STD_LOGIC_VECTOR (3 downto 0);
 signal clk2_to_wr : STD_LOGIC;
 signal data_out_temp : STD_LOGIC_VECTOR (15 downto 0);
 signal logic_high : STD_LOGIC_VECTOR (3 downto 0):="1111";

begin

 data_out(15 downto 0) <= data_out_temp(15 downto 0);

 m_pr_module: pr_module
 port map (
  reset => reset,
  clk1 => clk1,
  clk2 => clk2_to_wr,
  control_mode =>control_mode_wr,
  framer_ctl => framer_ctl_wr,
  pr_module_out => data_out_temp);

 -- freeze all inputs

control_mode_wr <= logic_high when (freeze ='1') else control_mode;

framer_ctl_wr <= logic_high when (freeze ='1') else framer_ctl;

clk2_to_wr <= logic_high(0) when (freeze ='1') else clk2;

end architecture;
```

Clocks and Other Global Signals for a PR Design

For non-PR designs, the Quartus II software automatically promotes high fan-out signals onto dedicated clocks or other forms of global signals during the pre-fitter stage of design compilation using a process called global promotion. For PR designs, however, automatic global promotion is disabled by default for PR regions, and you must assign the global clock resources necessary for PR partitions. Clock resources can be assigned by making Global Signal assignments in the Quartus II Assignment Editor, or by adding Clock Control Block (altclkctrl) Megafunction blocks in the design that drive the desired global signals.

There are 16 global clock networks in a Stratix V device. However, only six unique clocks can drive a row clock region limiting you to a maximum of six global signals in each PR region. The Quartus II software must ensure that any global clock can feed every location in the PR region.

The limit of six global signals to a PR region includes the `GCLK`, `QCLK` and `PCLKs` used inside of the PR region. Make QSF assignments for global signals in your project's Quartus II Settings File (**.qsf**), based on the clocking requirements for your design. In designs with multiple clocks that are external to the PR region, it may be beneficial to align the PR region boundaries to be within the global clock boundary (such as `QCLK` or `PCLK`).

If your PR region requires more than six global signals, modify the region architecture to reduce the number of global signals within this to six or fewer. For example, you can split a PR region into multiple regions, each of which uses only a subset of the clock domains, so that each region does not use more than six.

Every instance of a PR region that uses the global signals (for example, `PCLK`, `QCLK`, `GCLK`, `ACLR`) must use a global signal for that input.

Global signals can only be used to route certain secondary signals into a PR region and the restrictions for each block are listed in the following table. Data signals and other secondary signals not listed in the table, such as synchronous clears and clock enables are not supported.

Table 4-2: Supported Signal Types for Driving Clock Networks in a PR Region

Block Types	Supported Signals for Global/Periphery/Quadrant Clock Networks
LAB	`Clock, ACLR`
RAM	`Clock, ACLR, Write Enable(WE), Read Enable(RE)`
DSP	`Clock, ACLR`

Note: PR regions are allowed to contain output ports that are used outside of the PR region as global signals.

- If a global signal feeds both static and reconfigurable logic, the restrictions in the table also apply to destinations in the static region. For example, the same global signal cannot be used as an `SCLR` in the static region and an `ACLR` in the PR region.
- A global signal used for a PR region should only feed core blocks inside and outside the PR region. In particular you should not use a clock source for a PR region and additionally connect the signal to an I/O register on the top or bottom of the device. Doing so may cause the Assembler to give an error because it is unable to create valid programming mask files.

Floorplan Assignments for PR Designs

You must create a LogicLock region so the interface of the PR region with the static region is the same for any persona you implement. If different personas of a PR region have different area requirements, you must make a LogicLock region assignment that contains enough resources to fit the largest persona for the region. The static regions in your project do not necessarily require a floorplan, but depending on any other design requirement, you may choose to create a floorplan for a specific static region. If you create multiple PR regions, and are using SCRUB mode, make sure you have one column or row of static region between each PR region.

There is no minimum or maximum size for the LogicLock region assigned for a PR region. Because wire-LUTs are added on the periphery of a PR region by the Quartus II software, the LogicLock region for a PR region must be slightly larger than an equivalent non-PR region. Make sure the PR regions include only the resources that can be partially reconfigured; LogicLock regions for PR can only contain only LABs, DSPs, and RAM blocks. When creating multiple PR regions, make sure there is at least one static region column between each PR region. When multiple PR regions are present in a design, the shape and alignment of the region determines whether you use the SCRUB or AND/OR PR mode.

You can use the default **Auto size** and **Floating location** LogicLock region properties to estimate the preliminary size and location for the PR region.

You can also define regions in the floorplan that match the general location and size of the logic in each partition. You may choose to create a LogicLock region assignment that is non-rectangular, depending on the design requirements, but disjoint LogicLock regions are not allowed for PR regions.

After compilation, use the Fitter-determined size and origin location as a starting point for your design floorplan. Check the quality of results obtained for your floorplan location assignments and make changes to the regions as needed.

Alternatively, you can perform Analysis and Synthesis, and then set the regions to the required size based on resource estimates. In this case, use your knowledge of the connections between partitions to place the regions in the floorplan.

For more information on making design partitions and using an incremental design flow, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Floorplan Design* chapter in the *Quartus II Handbook*. For more design guidelines to ensure good quality of results, and suggestions on making design floorplan assignments with LogicLock regions, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Floorplan Assignments* chapter in the *Quartus II Handbook*.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Floorplan** on page 3-1
- **Best Practices for Incremental Compilation Partitions and Floorplan** on page 14-1

Implementation Details for Partial Reconfiguration

This section describes implementation details that help you create your PR design.

Partial Reconfiguration Pins

Partial reconfiguration can be performed through external pins or from inside the core of the FPGA.

When using PR from pins, some of the I/O pins are dedicated for implementing partial reconfiguration functionality. If you perform partial reconfiguration from pins, then you must use the passive parallel with 16 data bits (FPPx16) configuration mode. All dual-purpose pins should also be specified to **Use as regular I/O**.

To enable partial reconfiguration from pins in the Quartus II software, perform the following steps:

1. From the Assignments menu, click **Device**, then click **Device and Pin Options**.
2. In the **Device and Pin Options** dialog box, select **General** in the **Category** list and turn on **Enable PR pins** from the **Options** list.
3. Click **Configuration** in the **Category** list and select **Passive Parallel x16** from the **Configuration scheme** list.
4. Click **Dual-Purpose Pins** in the **Category** list and verify that all pins are set to **Use as regular I/O** rather than **Use as input tri-stated**.
5. Click **OK**, or continue to modify other settings in the **Device and Pin Options** dialog box.
6. Click **OK**.

Note: You can enable open drain on PR pins from the **Device and Pin Options** dialog box in the **Settings** page of the Quartus II software.

Table 4-3: Partial Reconfiguration Dedicated Pins Description

Pin Name	Pin Type	Pin Description
`PR_REQUEST`	Input	Dedicated input when **Enable PR pins** is turned on; otherwise, available as user I/O. Logic high on pin indicates the PR host is requesting partial reconfiguration.
`PR_READY`	Output	Dedicated output when **Enable PR pins** is turned on; otherwise, available as user I/O. Logic high on this pin indicates the Stratix V control block is ready to begin partial reconfiguration.
`PR_DONE`	Output	Dedicated output when **Enable PR pins** is turned on; otherwise, available as user I/O. Logic high on this pin indicates that partial reconfiguration is complete.
`PR_ERROR`	Output	Dedicated output when **Enable PR pins** is turned on; otherwise, available as user I/O. Logic high on this pin indicates the device has encountered an error during partial reconfiguration.

Pin Name	Pin Type	Pin Description
`DATA[15:0]`	Input	Dedicated input when **Enable PR pins** is turned on; otherwise available as user I/O. These pins provide connectivity for `PR_DATA` when **Enable PR pins** is turned on.
`DCLK`	Bidirectional	Dedicated input when **Enable PR pins** is turned on; `PR_DATA` is sent synchronous to this clock. This is a dedicated programming pin, and is not available as user I/O even if **Enable PR pins** is turned off.

For more information on different configuration modes for Stratix V devices, and specifically about FPPx16 mode, refer to the *Configuration, Design Security, and Remote System Upgrades in Stratix V Devices* chapter of the *Stratix V Handbook*.

Related Information

Configuration, Design Security, and Remote System Upgrades in Stratix V Devices

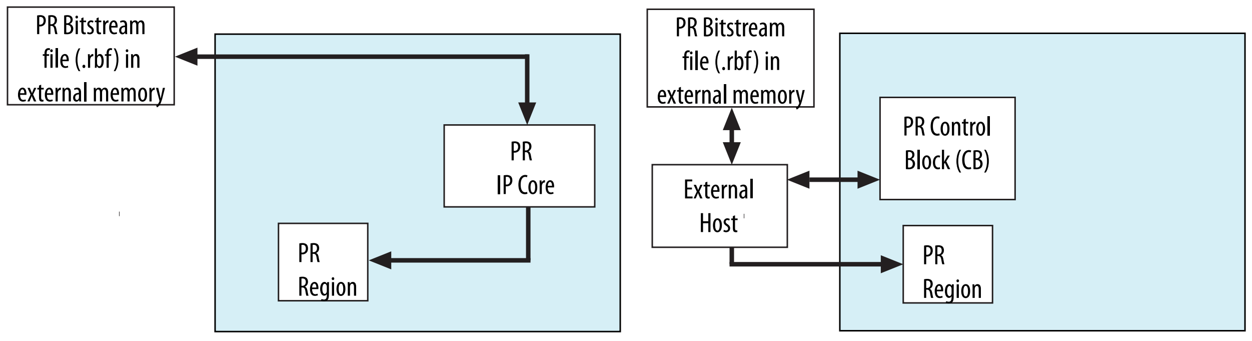
Interface with the PR Control Block through a PR Host

You communicate between your PR control IP and the PR Control Block (CB) via control signals, while executing partial reconfiguration.

You can communicate with the PR control block via an internal host which communicates with the CB through internal control signals. You can also use an external host with handshake signals accessed via external pins. The internal PR host can be user logic or a Nios® II processor.

Figure 4-9: Managing Partial Reconfiguration with an Internal or External Host

The figure shows how these blocks should be connected to the PR control block (CB). In your system, you will have either the External Host or the Internal Host, but not both.


PR Bitstream file (.rbf) in external memory
PR IP Core
PR Region
PR Bitstream file (.rbf) in external memory
External Host
PR Control Block (CB)
PR Region

The PR mode is independent of the full chip programming mode. For example, you can configure the full chip using a JTAG download cable, or other supported configuration modes. When configuring PR regions, you must use the FPPx16 interface to the PR control block whether you choose to partially reconfigure the chip from an external or internal host.

When using an external host, you must implement the control logic for managing system aspects of partial reconfiguration on an external device. By using an internal host, you can implement all of your logic necessary for partial reconfiguration in the FPGA, therefore external devices are not required to support partial reconfiguration. When using an internal host, you can use any interface to load the PR bitstream data to the FPGA, for example, from a serial or a parallel flash device, and then format the PR bitstream data to fit the FPPx16 interface on the PR Control Block.

To use the external host for your design, turn on the **Enable PR Pins** option in the **Device and Pin Options** dialog box in the Quartus II software when you compile your design. If this setting is turned off, then you must use an internal host. Also, you must tie the `corectl` port on the PR control block instance in the top-level of the design to the appropriate level for the selected mode.

Related Information

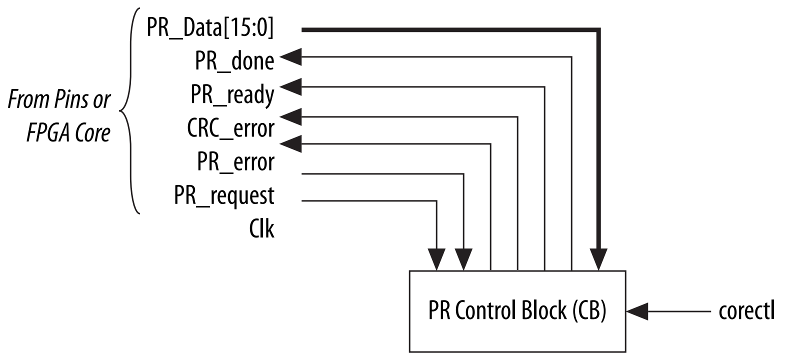
Partial Reconfiguration Pins on page 4-19
Partial Reconfiguration Dedicated Pins Table

PR Control Signals Interface

The Quartus II Programmer allows you to generate the different bit-streams necessary for full chip configuration and for partial reconfiguration. The programming bit-stream for partial reconfiguration contains the instructions (opcodes) as well as the configuration bits, necessary for reconfiguring each of the partial regions. When using an external host, the interface ports on the control block are mapped to FPGA pins. When using an internal host, these signals are within the core of the FPGA.

Figure 4-10: Partial Reconfiguration Interface Signals

These handshaking control signals are used for partial reconfiguration.



- `PR_DATA`: The configuration bitstream is sent on `PR_ DATA[ 15:0]`, synchronous to the `Clk`.
- `PR_DONE`: Sent from CB to control logic indicating the PR process is complete.
- `PR_READY`: Sent from CB to control logic indicating the CB is ready to accept PR data from the control logic.
- `CRC_Error`: The CRC_Error generated from the device's CRC block, is used to determine whether to partially reconfigure a region again, when encountering a CRC_Error.

- PR_ERROR: Sent from CB to control logic indicating an error during partial reconfiguration.
- PR_REQUEST: Sent from your control logic to CB indicating readiness to begin the PR process.
- corectl: Determines whether partial reconfiguration is performed internally or through pins.

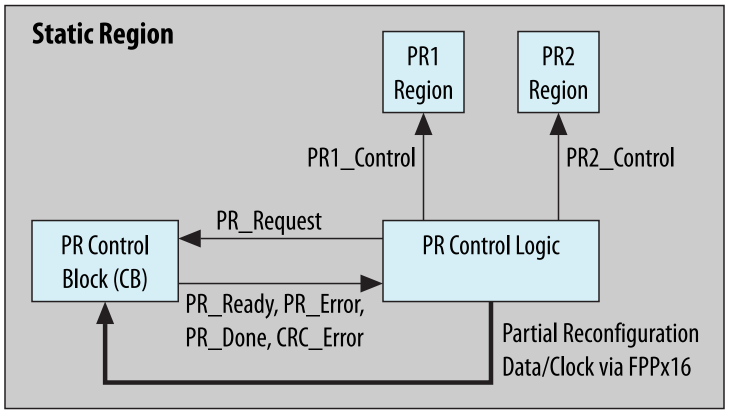
Reconfiguring a PR Region

The figure below shows a system in which your PR Control logic is implemented inside the FPGA. However, this section is also applicable for partial reconfiguration with an external host.

The PR control block (CB) represents the Stratix V PR controller inside the FPGA. PR1 and PR2 are two PR regions in a user design. In addition to the four control signals (PR_REQUEST, PR_READY, PR_DONE, PR_ERROR) and the data/clock signals interfacing with the PR control block, your PR Control IP should also send a control signal (PR_CONTROL) to each PR region. This signal implements the freezing and unfreezing of the PR Interface signals. This is necessary to avoid contention on the FPGA routing fabric.

Figure 4-11: Example of a PR System with Two PR Regions

Implementation of PR Control logic in the FPGA.



After the FPGA device has been configured with a full chip configuration at least once, the INIT_DONE signal is released, and the signal is asserted high due to the external resistor on this pin. The INIT_DONE signal must be assigned to a pin to monitor it externally. When a full chip configuration is complete, and the device is in user mode, the following steps describe the PR sequence:

1. Begin a partial reconfiguration process from your PR Control logic, which initiates the PR process for one or more of the PR regions (asserting PR1_Control or PR2_Control in the figure). The wrapper HDL described earlier freezes (pulls high) all non-global inputs of the PR region before the PR process.
2. Send PR_REQUEST signal from your control logic to the PR Control Block (CB). If your design uses an external controller, monitor INIT_DONE to verify that the chip is in user mode before asserting the PR_REQUEST signal. The CB initializes itself to accept the PR data and clock stream. After that, the CB asserts a PR_READY signal to indicate it can accept PR data. Exactly four clock cycles must occur before sending the PR data to make sure the PR process progresses correctly. Data and clock signals are sent to the PR control block to partially reconfigure the PR region interface.

- If there are multiple PR personas for the PR region, your PR Control IP must determine the programming file data for partial reconfiguration.
- When there are multiple PR regions in the design, then the same PR control IP determines which regions require reconfiguration based on system requirements.
- At the end of the PR process, the PR control block asserts a `PR_DONE` signal and de-asserts the `PR_READY` signal.
- If you want to suspend sending data, you can implement logic to pause the clock at any point.

3. Your PR control logic must de-assert the `PR_REQUEST` signal within eight clock cycles after the `PR_DONE` signal goes high. If your logic does not de-assert the `PR_REQUEST` signal within eight clock cycles, a new PR cycle starts.
4. If your design includes additional PR regions, repeat steps 2 – 3 for each region. Otherwise, proceed to step 5.
5. Your PR Control logic de-asserts the `PR_CONTROL` signal(s) to the PR region. The freeze wrapper releases all input signals of the PR region, thus the PR region is ready for normal user operation.
6. You must perform a reset cycle to the PR region to bring all logic in the region to a known state. After partial reconfiguration is complete for a PR region, the states in which the logic in the region come up is unknown.

The PR event is now complete, and you can resume operation of the FPGA with the newly configured PR region.

At any time after the start of a partial reconfiguration cycle, the PR host can suspend sending the `PR_DATA`, but the host must suspend sending the `PR_CLK` at the same time. If the `PR_CLK` is suspended after a PR process, there must be at least 20 clock cycles after the `PR_DONE` or `PR_ERROR` signal is asserted to prevent incorrect behavior.

For an overview of different reset schemes in Altera devices, refer to the *Recommended Design Practices* chapter in the *Quartus II Handbook*.

Related Information

Partial Reconfiguration Cycle Waveform on page 4-24
For more information on clock requirements for partial reconfiguration.

Recommended Design Practices on page 11-1

Partial Reconfiguration Cycle Waveform

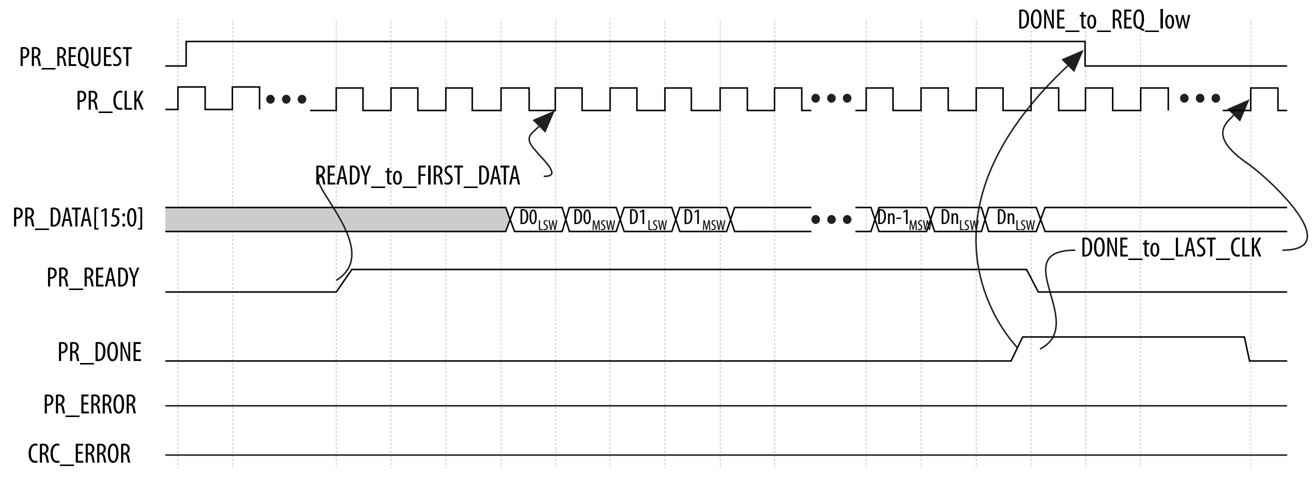
The PR host initiates the PR request, transfers the data to the FPGA device when it is ready, and monitors the PR process for any errors or until it is done.

Note: If you are using the PR IP core in the internal host mode, the IP core takes care of all the timing relationships mentioned below. As a user you do not have to do anything else. If you are using the PR IP core in external host mode, then you must pay attention to the timing relationships.

A PR cycle is initiated by the host (internal or external) by asserting the `PR_REQUEST` signal high. When the FPGA device is ready to begin partial reconfiguration, it responds by asserting the `PR_READY` signal high. The PR host responds by sending configuration data on `DATA [15:0]`. The data is sent synchronous to `PR_CLK`. When the FPGA device receives all PR data successfully, it asserts the `PR_DONE` high, and de-asserts `PR_READY` to indicate the completion of the PR cycle.

Figure 4-12: Partial Reconfiguration Timing Diagram

The partial reconfiguration cycle waveform with a hand-shaking protocol.



If there is an error encountered during partial reconfiguration, the FPGA device asserts the PR_ERROR signal high and de-asserts the PR_READY signal low.

The PR host must continuously monitor the PR_DONE and PR_ERROR signals status. Whenever either of these two signals are asserted, the host must de-assert PR_REQUEST within eight PR_CLK cycles. As a response to PR_ERROR error, the host can optionally request another partial reconfiguration or perform a full FPGA configuration.

To prevent incorrect behavior, the PR_CLK signal must be active a minimum of twenty clock cycles after PR_DONE or PR_ERROR signal is asserted high. Once PR_DONE is asserted, PR_REQUEST must be de-asserted within eight clock cycles. PR_DONE is de-asserted by the device within twenty PR_CLK cycles. The host can assert PR_REQUEST again after the 20 clocks after PR_DONE is de-asserted.

Table 4-4: Partial Reconfiguration Clock Requirements

Signal timing requirements for partial reconfiguration.

Timing Parameters	Value (clock cycles)
PR_READY to first data	4 (exact)
PR_ERROR to last clock	20 (minimum)
PR_DONE to last clock	20 (minimum)
DONE_to_REQ_low	8 (maximum)
Compressed PR_READY to first data	4 (exact)
Encrypted PR_READY to first data (when using double PR)	8 (exact)

Timing Parameters	Value (clock cycles)
Encrypted and Compressed PR_READY to first data (when using double PR)	12 (exact)

At any time during partial reconfiguration, to pause sending PR_DATA, the PR host can stop toggling PR_CLK. The clock can be stopped either high or low.

At any time during partial reconfiguration, the PR host can terminate the process by de-asserting the PR request. A partially completed PR process results in a PR error. You can have the PR host restart the PR process after a failed process by sending out a new PR request 20 cycles later.

If you terminate a PR process before completion, and follow it up with a full FPGA configuration by asserting nConfig, then you must toggle PR_CLK for an additional 20 clock cycles prior to asserting nConfig to flush the PR_CONTROL_BLOCK and avoid lock up.

During these steps, the PR control block might assert a PR_ERROR or a CRC_ERROR signal to indicate that there was an error during the partial reconfiguration process. Assertion of PR_ERROR indicates that the PR bitstream data was corrupt, and the assertion of CRC error indicates a CRAM CRC error either during or after completion of PR process. If the PR_ERROR or CRC_ERROR signals are asserted, you must plan whether to reconfigure the PR region or reconfigure the whole FPGA, or leave it unconfigured.

Important: The PR_CLK signal has different a nominal maximum frequency for each device. Most Stratix V devices have a nominal maximum frequency of at least 62.5 MHz.

Partial Reconfiguration with an External Host

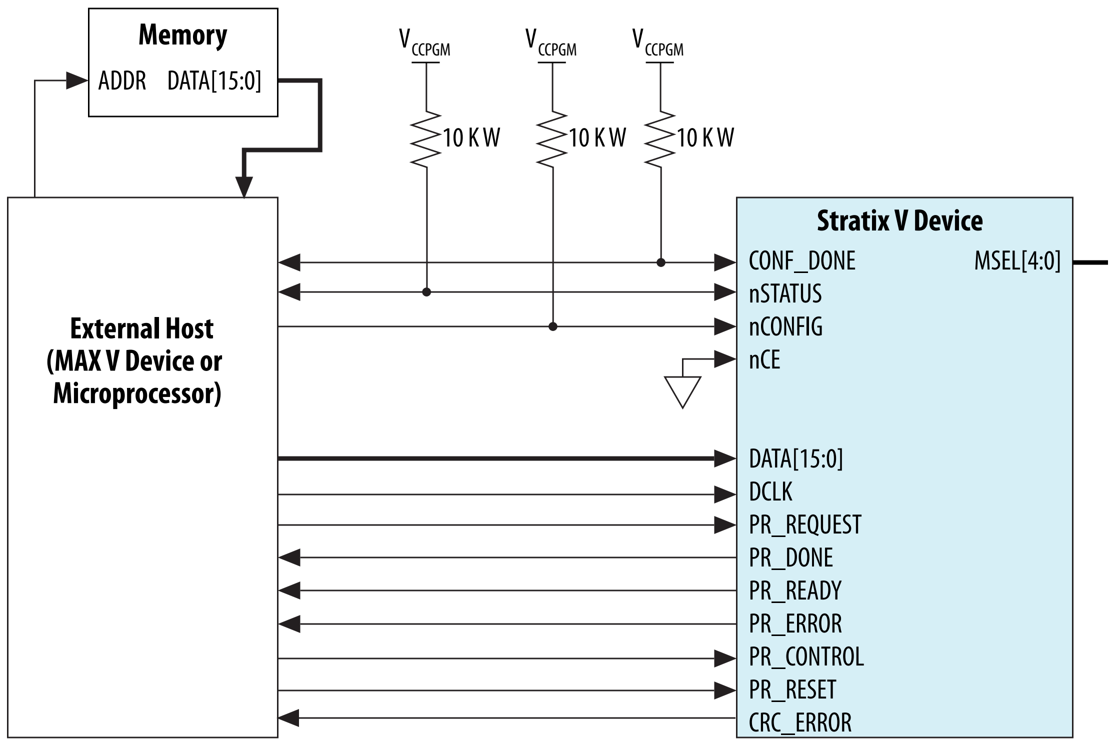
For partial reconfiguration using an external host, you must set the MSEL [4:0] pins for FPPx16 configuration scheme.

You can use a microcontroller, another FPGA, or a CPLD such as a MAX V device, to implement the configuration and PR controller. In this setup, the Stratix V device configures in FPPx16 mode during power-up. Alternatively, you can use a JTAG interface to configure the Stratix V device.

At any time during user-mode, the external host can initiate partial reconfiguration and monitor the status using the external PR dedicated pins: PR_REQUEST, PR_READY, PR_DONE, and PR_ERROR. In this mode, the external host must respond appropriately to the hand-shaking signals for a successful partial reconfiguration. This includes acquiring the data from the flash memory and loading it into the Stratix V device on DATA[15:0].

Figure 4-13: Connecting to an External Host

The connection setup for partial reconfiguration with an external host in the FPPx16 configuration scheme.



Note: If you don't care to write an external host controller, you implement an external host with the Partial Reconfiguration IP core on a MAX 10 or other FPGA device.

Related Information

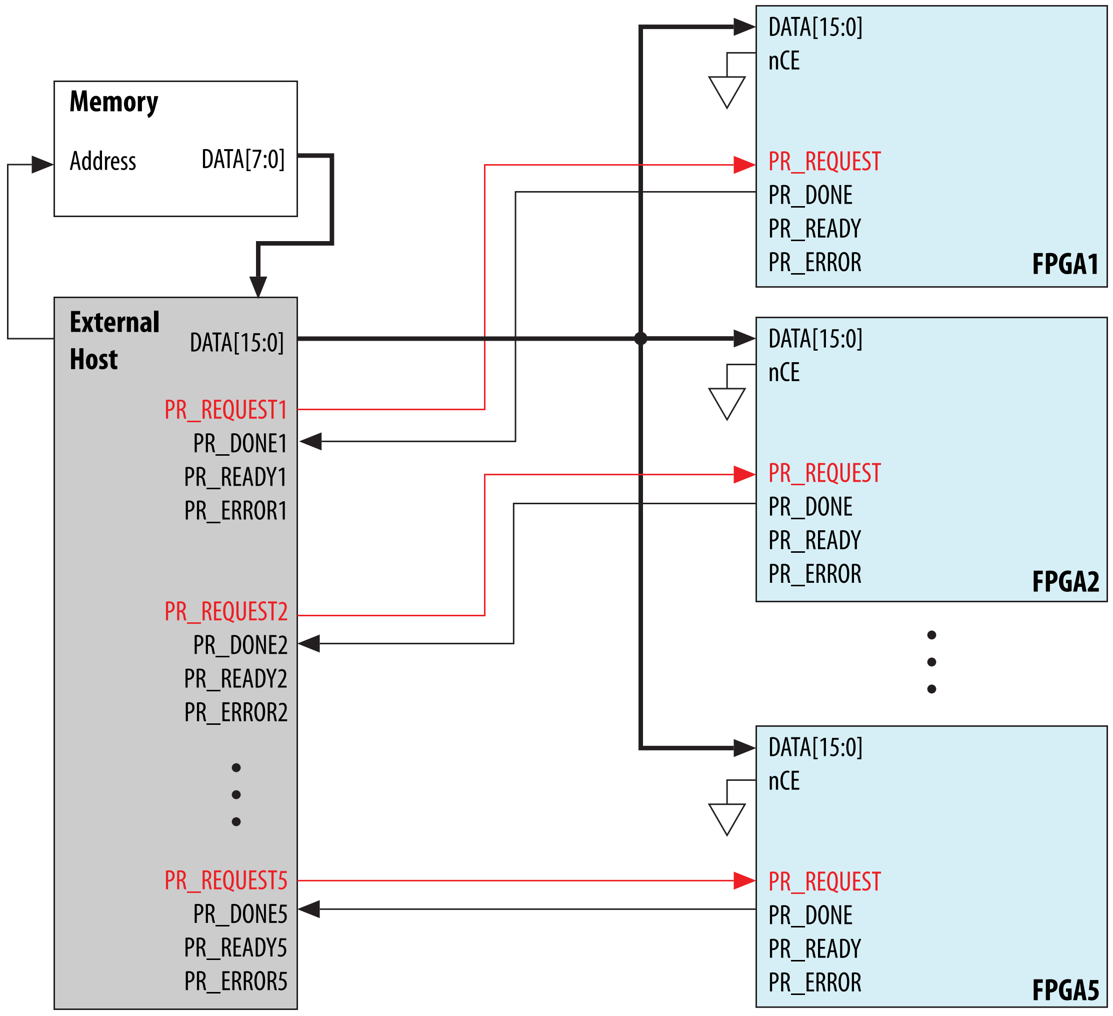
Partial Reconfiguration IP Core User Guide

Using an External Host with Multiple Devices

You must design the external host to accommodate the arbitration scheme that is required for your system, as well as the partial reconfiguration interface requirement for each device.

Figure 4-14: Connecting Multiple FPGAs to an External Host

An example of an external host controlling multiple Stratix V devices on a board.



Partial Reconfiguration with an Internal Host

You can create PR internal host logic with the PR IP core. If your design uses an internal host, the PR IP core handles the required hand-shaking protocol with the PR control block.

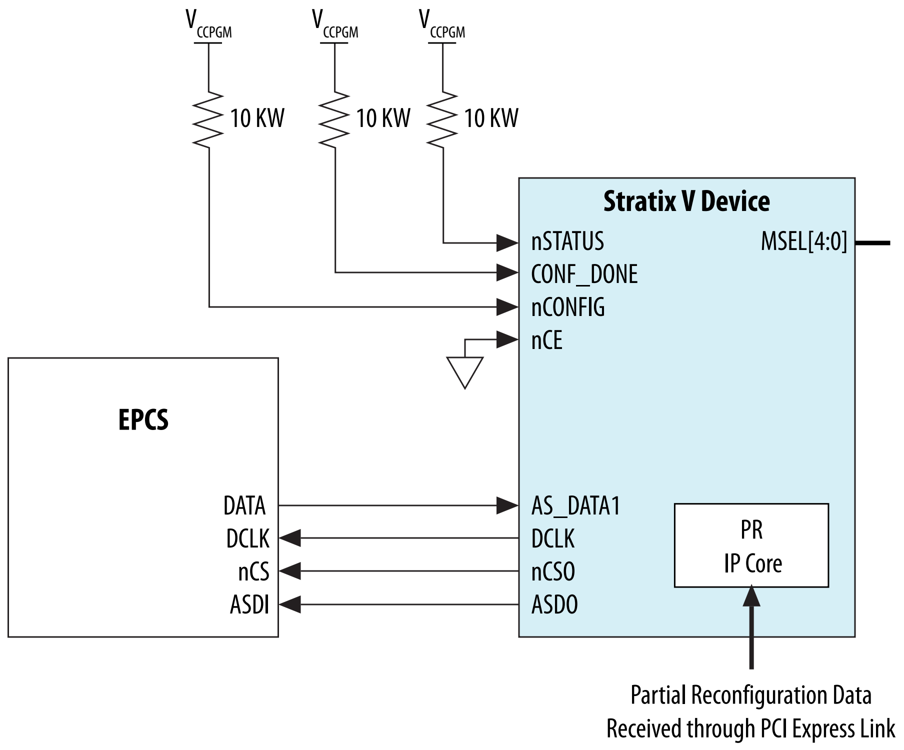
For example, PR programming bitstream(s) stored in an external flash device can be routed through the regular I/Os of the FPGA device, or received through the high speed transceiver channel (PCI Express, SRIO or Gigabit Ethernet), for processing by the internal logic.

The PR dedicated pins (`PR_REQUEST`, `PR_READY`, `PR_DONE`, and `PR_ERROR`) can be used as regular I/Os when performing partial reconfiguration with an internal host. For the full FPGA configuration upon power-up, you can set the `MSEL[4:0]` pins to match the configuration scheme, for example, AS, PS, FPPx8, FPPx16, or FPPx32. Alternatively, you can use the JTAG interface to configure the FPGA device. At any time during user-mode, you can initiate partial reconfiguration through the FPGA core fabric using the PR internal host.

In the following figure, the programming bitstream for partial reconfiguration is received through the PCI Express link, and your logic converts the data to the FPPx16 mode.

Figure 4-15: Connecting to an Internal Host

An example of the configuration setup when performing partial reconfiguration using the internal host.



Partial Reconfiguration Project Management

When compiling your PR project, you must create a base revision, and one or more reconfigurable revisions. The project revision you start out is termed the base revision.

Create Reconfigurable Revisions

To create a reconfigurable revision, use the **Revisions** tab of the **Project Navigator** window in the Quartus II software. When you create a reconfigurable revision, the Quartus II software adds the required assignments to associate the reconfigurable revision with the base revision of the PR project. You can add the necessary files to each revision with the **Add/Remove Files** option in the **Project** option under the **Project** menu in the Quartus II software. With this step, you can associate the right implementation files for each revision of the PR project.

Important: You must use the **Revisions** tab of the **Project Navigator** window in the Quartus II software when creating revisions for partial reconfiguration. Revisions created using **Project** > **Revisions** cannot be reconfigured.

Compiling Reconfigurable Revisions

Altera recommends that you use the largest persona of the PR region for the base compilation so that the Quartus II software can automatically budget sufficient routing.

Here are the typical steps involved in a PR design flow.

1. Compile the base revision with the largest persona for each PR region.
2. Create reconfigurable revisions for other personas of the PR regions by right-clicking in the **Revisions** tab in the Project Navigator.
3. Compile your reconfigurable revisions.
4. Analyze timing on each reconfigurable revision to make sure the design performs correctly to specifications.
5. Create aggregate revisions as needed.
6. Create programming files.

For more information on compiling a partial reconfiguration project, refer to *Performing Partial Reconfiguration* in Quartus II Help.

Related Information

Performing Partial Reconfiguration

Timing Closure for a Partial Reconfiguration Project

As with any other FPGA design project, simulate the functionality of various PR personas to make sure they perform to your system specifications. You must also make sure there are no timing violations in the implementation of any of the personas for every PR region in your design project.

In the Quartus II software, this process is manual, and you must run multiple timing analyses, on the base, reconfigurable, and aggregate revisions. The different timing requirements for each PR persona can be met by using different SDC constraints for each of the personas.

The interface between the partial and static partitions remains identical for each reconfigurable and aggregate revision in the PR flow. If all the interface signals between the static and the PR regions are registered, and there are no timing violations within the static region as well as within the PR regions, the reconfigurable and aggregate revisions should not have any timing violations.

However, you should perform timing analysis on the reconfigurable and aggregate revisions, in case you have any unregistered signals on the interface between partial reconfiguration and static regions.

Bitstream Compression and Encryption for PR Designs

You can choose to independently compress and encrypt the base bitstream as well as the PR bitstream for your PR project using options available in the Quartus II software.

When you choose to compress the bitstreams, you can compress the base and PR programming bitstreams independently, based on your design requirements. However, if you want to encrypt only the base image, you can choose wether or not to encrypt the PR images.

- When you want to encrypt the bitstreams, you can encrypt the PR images only when the base image is encrypted.
- The Encryption Key Programming (**.ekp**) file generated when encrypting the base image must be used for encrypting PR bitstream.
- When you compress the bitstream, you must present each `PR_DATA[15:0]` word for exactly four clock cycles.

Table 4-5: Partial Reconfiguration Clock Requirements for Bitstream Compression

Timing Parameters	Value (clock cycles)
`PR_READY` to first data	4 (exact)
`PR_ERROR` to last clock	80 (minimum)
`PR_DONE` to last clock	80 (minimum)
`DONE_to_REQ_low`	8 (maximum)

Related Information

- **Enable Partial Reconfiguration Bitstream Decompression when Configuring Base Design SOF file in JTAG mode** on page 4-36
- **Enable Bitstream Decryption Option** on page 4-37
- **Generate PR Programming Files with the Convert Programming Files Dialog Box** on page 4-34

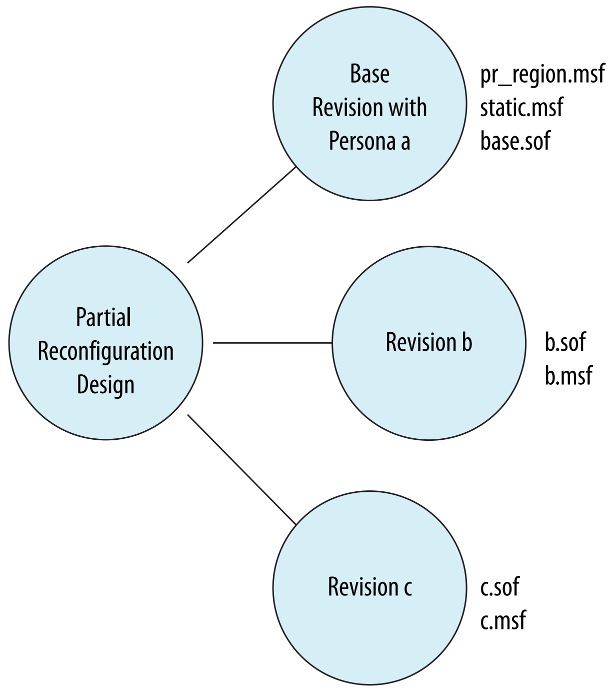
Programming Files for a Partial Reconfiguration Project

You must generate PR bitstream(s) based on the designs and send them to the control block for partial reconfiguration.

Compile the PR project, including the base revision and at least one reconfigurable revision before generating the PR bitstreams. The Quartus II Programmer generates PR bitstreams. This generated bitstream can be sent to the PR ports on the control block for partial reconfiguration.

Figure 4-16: PR Project with Three Revisions

Consider a partial reconfiguration design that has three revisions and one PR region, a base revision with persona a, one PR revision with persona b, and a second PR revision with persona c.

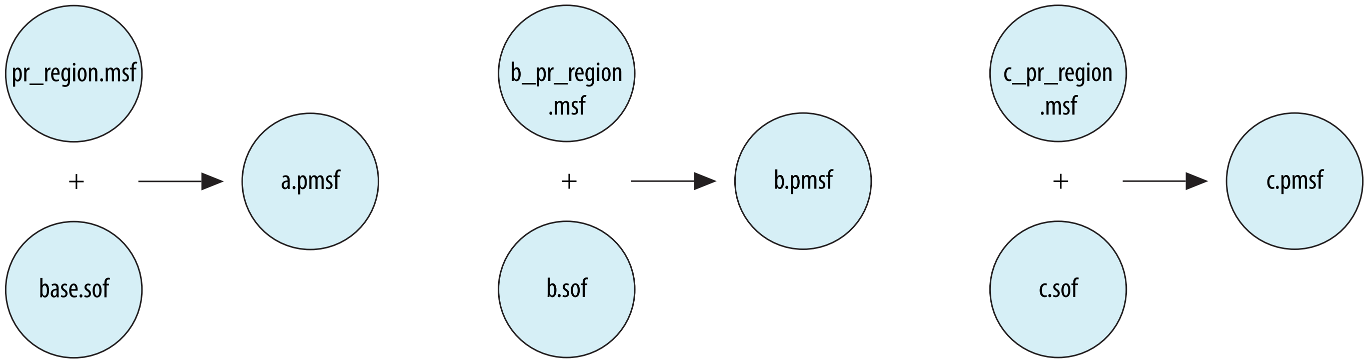


When these individual revisions are compiled in the Quartus II software, the assembler produces Masked SRAM Object Files (**.msf**) and the SRAM Object Files (**.sof**) for each revision. The **.sof** files are created as before (for non-PR designs). Additionally, **.msf** files are created specifically for partial reconfiguration, one for each revision. The **pr_region.mfsf** file is the one of interest for generating the PR bitstream. It contains the mask bits for the PR region. Similarly, the **static.msf** file has the mask bits for the static region. The **.sof** files have the information on how to configure the static region as well as the corresponding PR region. The **pr_region.msf** file is used to mask out the static region so that the bitstream can be computed for the PR region. The default file name of the pr region **.msf** corresponds to the LogicLock region name, unless the name is not alphanumeric. In the case of a non-alphanumeric region name, the **.msf** file is named after the location of the lower left most coordinate of the region.

Note: Altera recommends naming all LogicLock regions to enhance documenting your design.

Figure 4-17: Generation of Partial-Masked SRAM Object Files (.pmsf)

You can convert files in the Convert Programming Files window or run the `quartus_cpf -p` command to process the **pr_region.msf** and **.sof** files to generate the Partial-Masked SRAM Object File (**.pmsf**).



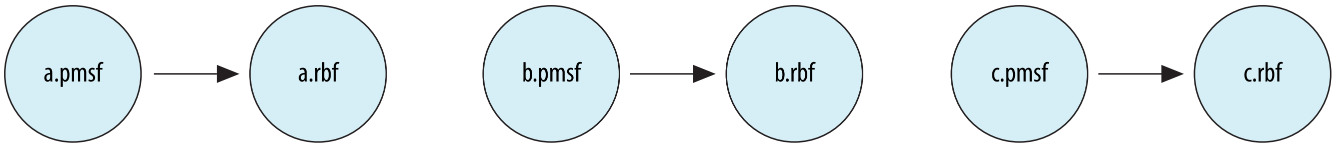
The **.msf** file helps determine the PR region from each of the **.sof** files during the PR bitstream computation.

Once all the **.pmsf** files are created, process the PR bitstreams by running the `quartus_cpf -o` command to produce the raw binary **.rbf** files for reconfiguration.

If one wishes to partially reconfigure the PR region with persona a, use the **a.rbf** bitstream file, and so on for the other personas.

Figure 4-18: Generating PR Bitstreams

This figure shows how three bitstreams can be created to partially reconfigure the region with persona a, persona b, or persona c as desired.



In the Quartus II software, the Convert Programming Files window supports the generation of the required programming bitstreams. When using the `quartus_cpf` from the command line, the following options for generating the programming files are read from an option text file, for example, **option.txt**.

- If you want to use SCRUB mode, before generating the bitstreams create an option text file, with the following line:

  ```
  use_scrub=on
  ```

- If you have initialized M20K blocks in the PR region (ROM/Initialized RAM), then add the following line in the option text file, before generating the bitstreams:

  ```
  write_block_memory_contents=on
  ```

- If you want to compress the programming bitstream files, add the following line in the option text file. This option is available when converting base **.sof** to any supported programming file types, such as **.rbf**, **.pof** and JTAG Indirect Configuration File (**.jic**).

  ```
  bitstream_compression=on
  ```

Related Information

Generate PR Programming Files with the Convert Programming Files Dialog Box on page 4-34

Generating Required Programming Files

1. Generate **.sof** and **.msf** files (part of a full compilation of the base and PR revisions).
2. Generate a Partial-Masked SRAM Object File (**.pmsf**) using the following commands:

   ```
   quartus_cpf -p <pr_revision>.msf <pr_revision>.sof <new_filename>.pmsf
   ```

   for example:

   ```
   quartus_cpf -p x7y48.msf switchPRBS.sof x7y48_new.pmsf
   ```
3. Convert the **.pmsf** file for every PR region in your design to **.rbf** file format. The **.rbf** format is used to store the bitstream in an external flash memory. This command should be run in the same directory where the files are located:

   ```
   quartus_cpf -o scrub.txt -c <pr_revision >.pmsf <pr_revision>.rbf
   ```

   for example:

   ```
   quartus_cpf -o scrub.txt -c x7y48_new.pmsf x7y48.rbf
   ```

When you do not have an option text file such as **scrub.txt**, the files generated would be for AND/OR mode of PR, rather than SCRUB mode.

Generate PR Programming Files with the Convert Programming Files Dialog Box

In the Quartus II software, the flow to generate PR programming files is supported in the Convert Programming Files dialog box. You can specify how the Quartus II software processes file types such as **.msf**, **.pmsf**, and **.sof** to create **.rbf** and merged **.msf** and **.pmsf** files.

You can create

- A **.pmsf** output file, from **.msf** and **.sof** input files
- A **.rbf** output file from a **.pmsf** input file
- A merged **.msf** file from two or more **.msf** input files
- A merged **.pmsf** file from two or more **.pmsf** input files

Convert Programming Files dialog box also allows you to enable the option bit for bitstream decompression during partial reconfiguration, when converting the base **.sof** (full design **.sof**) to any supported file type.

For additional details, refer to the *Quartus II Programmer* chapter in the *Quartus II Handbook*.

Related Information

Quartus II Programmer

Generating a .pmsf File from a .msf and .sof Input File

Perform the following steps in the Quartus II software to generate the **.pmsf** file in the **Convert Programming Files** dialog box.

1. Open the **Convert Programming Files** dialog box.
2. Specify the programming file type as **Partial-Masked SRAM Object File (.pmsf)**.
3. Specify the output file name.
4. Select input files to convert (only a single **.msf** and **.sof** file are allowed). Click **Add**.
5. Click **Generate** to generate the **.pmsf** file.

Generating a .rbf File from a .pmsf Input File

Perform the following steps in the Quartus II software to generate the partial reconfiguration **.rbf** file in the **Convert Programming Files** dialog box.

1. From the File menu, click **Convert Programming Files**.
2. Specify the programming file type as **Raw Binary File for Partial Reconfiguration (.rbf)**.
3. Specify the output file name.
4. Select input file to convert. Only a single **.pmsf** input file is allowed. Click **Add**.
5. Select the new **.pmsf** and click **Properties**.
6. Turn the **Compression**, **Enable SCRUB mode**, **Write memory contents**, and **Generate encrypted bitstream** options on or off depending on the requirements of your design. Click **Generate** to generate the **.rbf** file for partial reconfiguration.

- **Compression**: Enables compression on the PR bitstream.
- **Enable SCRUB mode**: Default is based on AND/OR mode. This option is valid only when your design does not contain vertically overlapped PR masks. The **.rbf** generation fails otherwise.
- **Write memory contents**: Turn this on when you have a **.mif** that was used during compilation. Otherwise, turning this option on forces you to use double PR in AND/OR mode.
- **Generate encrypted bitstream**: If this option is enabled, you must specify the Encrypted Key Programming (**.ekp**) file, which generated when converting a base **.sof** to an encrypted bitstream. The same **.ekp** must be used to encrypt the PR bitstream.

When you turn on **Compression**, you must present each `PR_DATA[15:0]` word for exactly four clock cycles.

Turn on the **Write memory contents** option only if you are using AND/OR mode and have M20K blocks in your PR design that need to be initialized. When you check this box, you must to perform double PR for regions with initialized M20K blocks.

Related Information

- **Initializing M20K Blocks with a Double PR Cycle** on page 4-43
- **Initializing M20K Blocks with a Double PR Cycle** on page 4-43

Create a Merged .msf File from Multiple .msf Files

You can merge two or more **.msf** files in the **Convert Programming Files** window.

1. Open the **Convert Programming Files** window.
2. Specify the programming file type as **Merged Mask Settings File (.msf)**.
3. Specify the output file name.
4. Select **MSF Data** in the **Input files to convert** window.
5. Click Add File to add input files. You must specify two or more files for merging.
6. Click **Generate**to generate the merged file.

To merge two or more **.msf** files from the command line, type:

```
quartus_cpf --merge_msf=<number of merged files> <msf_input_file_1>
<msf_input_file_2> <msf_input_file_etc> <msf_output_file>
```

For example, to merge two **.msf** files, type:

```
quartus_cpf --merge_msf=<2> <msf_input_file_1> <msf_input_file_2>
<msf_output_file>
```

Generating a Merged .pmsf File from Multiple .pmsf Files

You can merge two or more **.pmsf** files in the **Convert Programming Files** window.

1. Open the **Convert Programming Files** window.
2. Specify the programming file type as **Merged Partial-Mask SRAM Object File (.pmsf)**.
3. Specify the output file name.
4. Select **PMSF Data** in the **Input files to convert** window.
5. Click **Add File** to add input files. You must specify two or more files for merging.
6. Click **Generate** to generate the merged file.

To merge two or more **.pmsf** files from the command line, type:

```
quartus_cpf --merge_pmsf=<number of merged files> <pmsf_input_file_1>
<pmsf_input_file_2> <pmsf_input_file_etc> <pmsf_output_file>
```

For example, to merge two **.pmsf** files, type:

```
quartus_cpf --merge_pmsf=<2> <pmsf_input_file_1> <pmsf_input_file_2>
<pmsf_output_file>
```

The merge operation checks for any bit conflict on the input files, and the operation fails with error message if a bit conflict is detected. In most cases, a successful file merge operation indicates input files do not have any bit conflict.

Enable Partial Reconfiguration Bitstream Decompression when Configuring Base Design SOF file in JTAG mode

In the Quartus II software, the **Convert Programming Files** window provides the option in the **.sof** file properties to enable bitstream decompression during partial reconfiguration.

This option is available when converting base **.sof** to any supported programming file types, such as **.rbf**, **.pof**, and **.jic**.

In order to view this option, the base **.sof** must be targeted on Stratix V devices in the **.sof File Properties**. This option must be turned on if you turned on the **Compression** option during **.pmsf** to **.rbf** file generation.

Enable Bitstream Decryption Option

The **Convert Programming Files** window provides the option in the **.sof** file properties to enable bitstream decryption during partial reconfiguration.

This option is available when converting base **.sof** to any supported programming file types, such as **.rbf**, **.pof**, and **.jic**.

The base **.sof** must have partial reconfiguration enabled and the base **.sof** generated from a design that has a PR Control Block instantiated, to view this option in the **.sof File Properties**. This option must be turned on if you wants to turn on the Generate encrypted bitstream option during **.pmsf** to **.rbf** file generation.

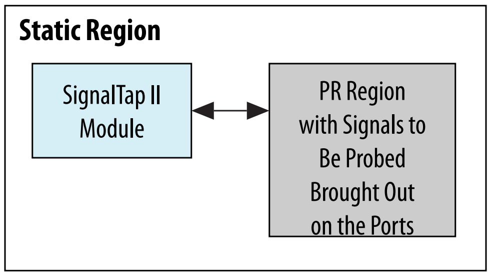
On-Chip Debug for PR Designs

You cannot instantiate a SignalTap II block inside a PR region. If you must monitor signals within a PR region for debug purposes, bring those signals to the ports of the PR region.

The Quartus II software does not support the Incremental SignalTap feature for PR designs. After you instantiate the SignalTap II block inside the static region, you must recompile your design. When you recompile your design, the static region may have a modified implementation and you must also recompile your PR revisions. If you modify an existing SignalTap II instance you must also recompile your entire design; base revision and reconfigurable revisions.

Figure 4-19: Using SignalTap II with a PR Design

You can instantiate the SignalTap II block in the static region of the design and probe the signals you want to monitor.


Static Region
SignalTap II Module
PR Region with Signals to Be Probed Brought Out on the Ports

You can use other on-chip debug features in the Quartus II software, such as the In-System Sources and Probes or SignalProbe, to debug a PR design. As in the case of SignalTap, In-System Sources and Probes can only be instantiated within the static region of a PR design. If you have to probe any signal inside the PR region, you must bring those signals to the ports of the PR region in order to monitor them within the static region of the design.

Partial Reconfiguration Known Limitations

There are restrictions that derive from hardware limitations in specific Stratix V devices.

The restrictions in the following sections apply only if your design uses M20K blocks as RAMs or ROMs in your PR project.

Memory Blocks Initialization Requirement for PR Designs

For a non-PR design, the power up value for the contents of a M20K RAM or a MLAB RAM are all set at zero. However, at the end of performing a partial reconfiguration, the contents of a M20K or MLAB memory block are unknown. You must intentionally initialize the contents of all the memory to zero, if required by the functionality of the design, and not rely upon the power on values.

M20K RAM Blocks in PR Designs

When your PR design uses M20K RAM blocks in Stratix V devices, there are some restrictions which limit how you utilize the respective memory blocks as ROMs or as RAMs with initial content.

Related Information

Implementing Memories with Initialized Content on page 4-41

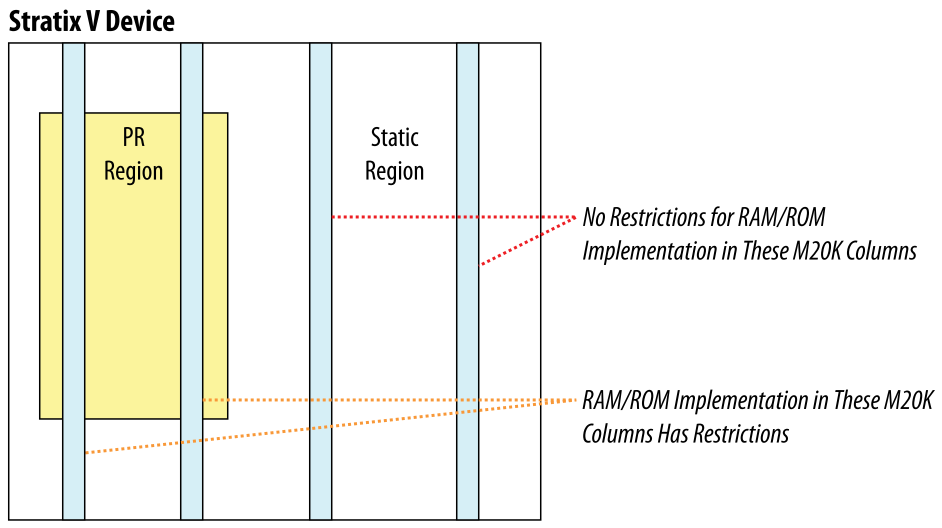
If your design requires initialized memory content either as a ROM or a RAM inside a PR region, you must follow these guidelines.

Limitations When Using Stratix V Production Devices

These workarounds allow your design to use M20K blocks with PR.

Figure 4-20: Limitations for Using M20Ks in PR Regions

If you implement a M20K block in your PR region as a ROM or a RAM with initialized content, when the PR region is reconfigured, any data read from the memory blocks in static regions in columns that cross the PR region is incorrect.


Stratix V Device
PR Region
Static Region
No Restrictions for RAM/ROM Implementation in These M20K Columns
RAM/ROM Implementation in These M20K Columns Has Restrictions

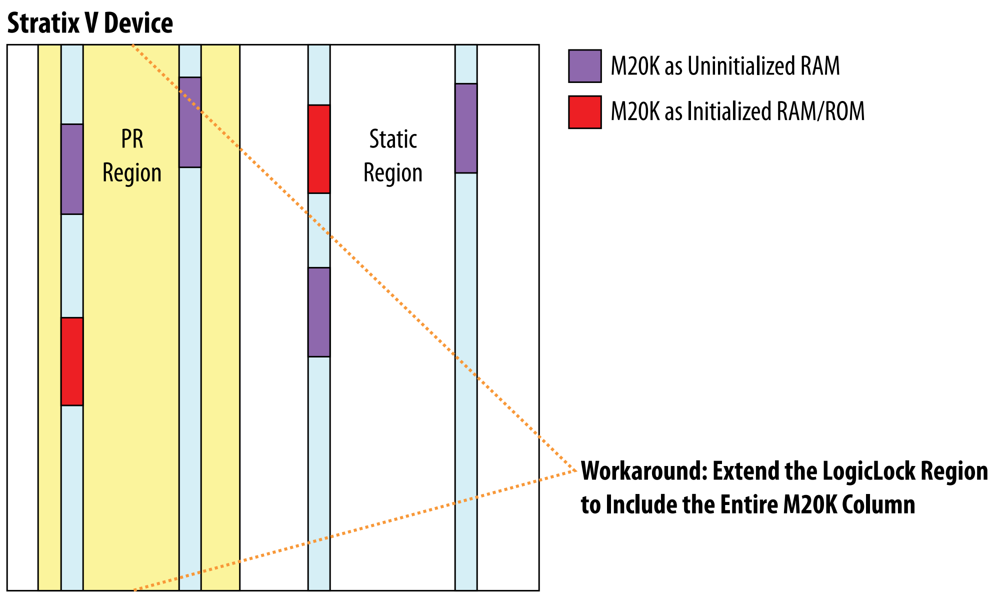
If the functionality of the static region depends on any data read out from M20K RAMs in the static region, the design will malfunction.

Use one of the following workarounds, which are applicable to both AND/OR and SCRUB modes of partial reconfiguration:

- Do not use ROMs or RAMs with initialized content inside PR regions.
- If this is not possible for your design, you can program the memory content for M20K blocks with a **.mif** using the suggested workarounds.
- Make sure your PR region extends vertically all the way through the device, in such a way that the M20K column lies entirely inside a PR region.

Figure 4-21: Workaround for Using M20Ks in PR Regions

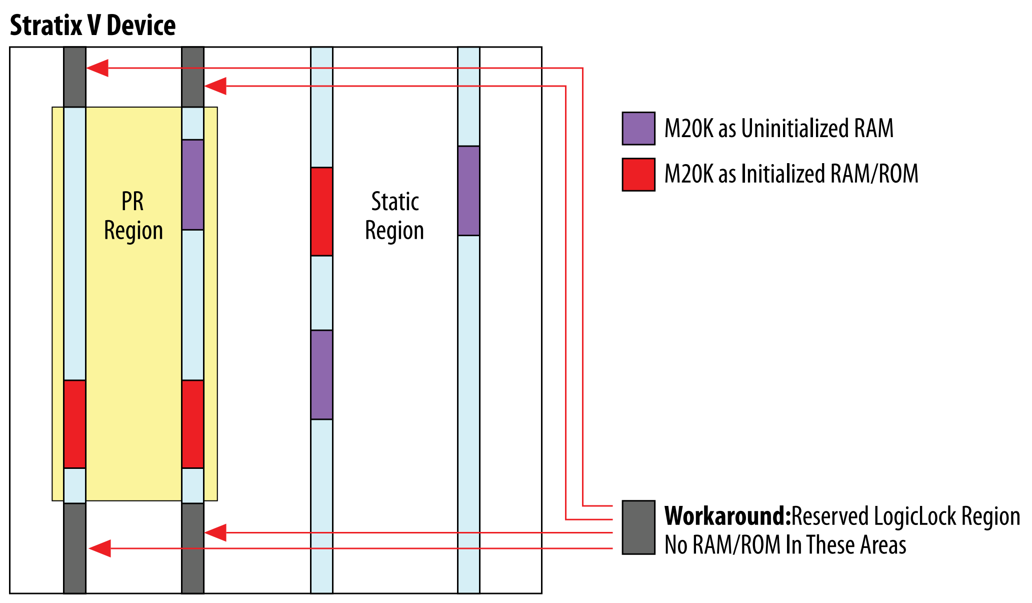
This figure shows the LogicLock region extended as a rectangle reducing the area available for the static region. However, you can create non-rectangular LogicLock regions to allocate the resources required for the partition more optimally. If saving area is a concern, extend the LogicLock region to include M20K columns entirely.


Stratix V Device
PR Region
Static Region
M20K as Uninitialized RAM
M20K as Initialized RAM/ROM
Workaround: Extend the LogicLock Region to Include the Entire M20K Column

•

Figure 4-22: Alternative Workaround for Using M20Ks in PR Region

Using Reserved LogicLock Regions, block all the M20K columns that are not inside a PR region, but that are in columns above or below a PR region. In this case, you may choose to under-utilize M20K resources, in order to gain ROM functionality within the PR region.


Stratix V Device
PR Region
Static Region
M20K as Uninitialized RAM
M20K as Initialized RAM/ROM
Workaround: Reserved LogicLock Region
No RAM/ROM In These Areas

For more information including a list of the Stratix V production devices, refer to the *Errata Sheet for Stratix V Devices*.

Related Information

Errata Sheet for Stratix V Devices

MLAB Blocks in PR designs

Stratix V devices include dual-purpose blocks called MLABs, which can be used to implement RAMs or LABs for user logic.

This section describes the restrictions while using MLAB blocks (sometimes also referred to as LUT-RAM) in Stratix V devices for your PR designs.

If your design uses MLABS as LUT RAM, you must use all available MLAB bits within the region.

Table 4-6: RAM Implementation Restrictions Summary

The following table shows a summary of the LUT-RAM Restrictions.

PR Mode	Type of memory in PR region	Stratix V Production
SCRUB mode	LUT RAM (no initial content)	OK
	LUT ROM and LUT RAM with your initial content	OK

PR Mode	Type of memory in PR region	Stratix V Production
AND/OR mode	LUT RAM (no initial content)	*While design is running:* Write 1s to all locations before partial reconfiguration *At compile time:* Explicitly initialize all memory locations in each new persona to 1 via initialization file (**. mif**).
	LUT ROM and LUT RAM with your initial content	No

If your design does not use any MLAB blocks as RAMs, the following discussion does not apply. The restrictions listed below are the result of hardware limitations in specific devices.

Limitations with Stratix V Production Devices

When using SCRUB mode:

- LUT-RAMs without initialized content, LUT-RAMs with initialized content, and LUT-ROMs can be implemented in MLABs within PR regions without any restriction.

When using AND/OR mode:

- LUT-RAMs with initialized content or LUT-ROMs cannot be implemented in a PR region.
- LUT-RAMs without initialized content in MLABs inside PR regions are supported with the following restrictions.
- MLAB blocks contain 640 bits of memory. The LUT RAMs in PR regions in your design must occupy all MLAB bits, you should not use partial MLABs.
- You must include control logic in your design with which you can write to all MLAB locations used inside PR region.
- Using this control logic, write '1' at each MLAB RAM bit location in the PR region before starting the PR process. This is to work around a false EDCRC error during partial reconfiguration.
- You must also specify a **.mif** that sets all MLAB RAM bits to '1' immediately after PR is complete.
- ROMs cannot be implemented in MLABs (LUT-ROMs).
- There are no restrictions to using MLABs in the static region of your PR design.

For more information, refer to the following documents in the *Stratix V Handbook*:

Related Information

Errata Sheet for Stratix V Devices

Implementing Memories with Initialized Content

If your Stratix V PR design implements ROMs, RAMs with initialization, or ROMs within the PR regions, using either M20K blocks or LUT-RAMs, then you must follow the following design guidelines to determine what is applicable in your case.

Table 4-7: Implementing Memory with Initialized Content in PR Designs

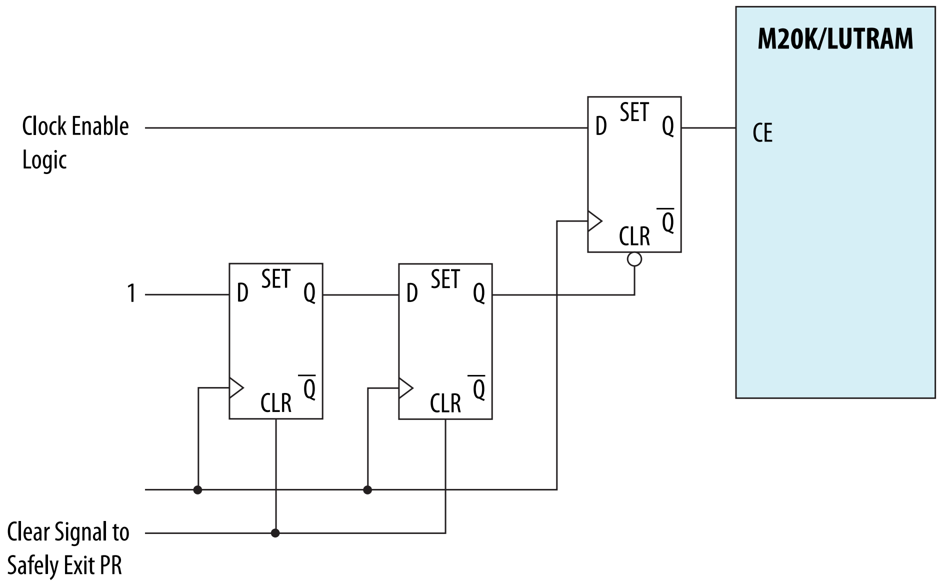
Mode		Production Devices	
		AND/OR	SCRUB
LUT-RAM without initialization	**Suggested Method**	*While design is running:* Write ‘1’ to all locations before partial reconfiguration. *At compile time:* Explicitly initialize all memory locations in each new persona to ‘1’ via initialization file (**.mif**) Make sure no spurious write on PR entry [4]	No special method required
	Without Suggested Method	CRC Error	No special method required
LUT-RAM with initialization	**Suggested Method**	Not supported	Make sure no spurious write on PR exit [4]
	Without Suggested Method		Incorrect results
M20K without initialization	**Suggested Method**	No special method required	
	Without Suggested Method	No special method required	
M20K with initialization	**Suggested Method**	Use double PR cycle [5] Make sure no spurious write on PR exit [4]	No special method required
	Without Suggested Method	Incorrect results	No special method required

(4) Use the circuit shown in the **M20K/LUTRAM** figure to create clock enable logic to safely exit partial reconfiguration without spurious writes.

(5) Double partial reconfiguration is described in *Initializing M20K Blocks with a Double PR Cycle*

Figure 4-23: M20K/LUTRAM

To avoid spurious writes during PR entry and exit, implement the following clock enable circuit in the same PR region as the RAM.



The circuit depends on an active- high clear signal from the static region. Before entering PR, freeze this signal in the same manner as all PR inputs. Your host control logic should de-assert the clear signal as the final step in the PR process.

Related Information

Initializing M20K Blocks with a Double PR Cycle on page 4-43

Initializing M20K Blocks with a Double PR Cycle

When a PR region in your PR design contains an initialized M20K block and is reconfigured via AND/OR mode, your host logic must complete a double PR cycle, instead of a single PR cycle.

The PR IP has a `double_pr` input port, that must be asserted high when your PR region contains RAM blocks that must be initialized. The PR IP core handles the timing relations between the first and the second PR cycles of a Double PR operation. From your user logic, assert the `double_pr` signal when you assert the `pr_start` signal, and you deassert the `double_pr` signal when the `freeze` signal is deasserted by the PR IP. This method is also applicable in cases when the programming bitstream is compressed or encryted.

Document Revision History

Table 4-8: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• Correct Verilog HDL partial reconfiguration instantiation code example. • Added clear/set method to SCRUB mode option.
2015.12.15	14.1.0	Minor revisions to some topics to resolve design refinements: • Implementing Memories with Initialized Content • Instantiating the PR Control Block and CRC Block in Verilog HDL • Partial Reconfiguration Pins
June 2014	14.0.0	Minor updates to "Programming File Sizes for a Partial Reconfiguration Project" and code samples in "Freeze Logic for PR Regions" sections.
November 2013	13.1.0	Added support for merging multiple .msf and .pmsf files. Added support for PR Megafunction. Updated for revisions on timing requirements.
May 2013	13.0.0	Added support for encrypted bitstreams. Updated support for double PR.
November 2012	12.1.0	Initial release.

Related Information

Quartus II Handbook Archive

For previous versions of the *Quartus II Handbook*.

Creating a System With Qsys 5

2015.05.04

QII5V1 Subscribe Send Feedback

Qsys is a system integration tool included as part of the Quartus II software. Qsys captures system-level hardware designs at a high level of abstraction and simplifies the task of defining and integrating customized IP components.

Qsys omponents include verification IP cores, and other design modules. Qsys facilitates design reuse by packaging and integrating your custom IP components with AlteraAltera and third-party IP components. Qsys automatically creates interconnect logic from the high-level connectivity that you specify, thereby eliminating the error-prone and time-consuming task of writing HDL to specify system-level connections.

Qsys is more powerful if you design your custom IP components using standard interfaces. By using standard interfaces, your custom IP components inter-operate with the Altera IP components in the IP Catalog. In addition, you can take advantage of bus functional models (BFMs), monitors, and other verification IP to verify your system.

Qsys supports Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), AMBA AXI4-Lite™ (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB™3 (version 1.0) interface specifications.

Qsys provides the following advantages:

- Simplifies the process of customizing and integrating IP components into systems
- Generates an IP core variation for use in your Quartus II software projects
- Supports up to 64-bit addressing
- Supports modular system design
- Supports visualization of systems
- Supports optimization of interconnect and pipelining within the system
- Supports auto-adaptation of different data widths and burst characteristics
- Supports inter-operation between standard protocols, such as Avalon and AXI
- Fully integrated with the Quartus II software

Note: For information on how to define and generate single IP cores for use in your Quartus II software projects, refer to *Introduction to Altera IP Cores* and *Managing Quartus II Projects*.

Related Information

- **Introduction to Altera IP Cores**
- **Managing Quartus II Projects** on page 1-1
- **Avalon Interface Specifications**
- **AMBA Protocol Specifications**

IP Component Interface Support in Qsys

IP components can have any number of interfaces in any combination. Each interface represents a set of signals that you can connect within a Qsys system, or export outside of a Qsys system.

Qsys IP components can include the following interface types:

Table 5-1: IP Component Interface Types

Interface Type	Description
Memory-Mapped	Connects memory-referencing master devices with slave memory devices. Master devices may be processors and DMAs, while slave memory devices may be RAMs, ROMs, and control registers. Data transfers between master and slave may be uni-directional (read only or write only), or bi-directional (read and write).
Streaming	Connects Avalon Streaming (Avalon-ST) sources and sinks that stream unidirectional data, as well as high-bandwidth, low-latency IP components. Streaming creates datapaths for unidirectional traffic, including multichannel streams, packets, and DSP data. The Avalon-ST interconnect is flexible and can implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet, Interlaken, and video. You can define bus widths, packets, and error conditions.
Interrupts	Connects interrupt senders to interrupt receivers. Qsys supports individual, single-bit interrupt requests (IRQs). In the event that multiple senders assert their IRQs simultaneously, the receiver logic (typically under software control) determines which IRQ has highest priority, then responds appropriately
Clocks	Connects clock output interfaces with clock input interfaces. Clock outputs can fan-out without the use of a bridge. A bridge is required only when a clock from an external (exported) source connects internally to more than one source.
Resets	Connects reset sources with reset input interfaces. If your system requires a particular positive-edge or negative-edge synchronized reset, Qsys inserts a reset controller to create the appropriate reset signal. If you design a system with multiple reset inputs, the reset controller ORs all reset inputs and generates a single reset output.
Conduits	Connects point-to-point conduit interfaces, or represent signals that are exported from the Qsys system. Qsys uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Qsys system as a point-to-point connection, or conduit interfaces can be exported and brought to the top-level of the system as top-level system I/O. You can use conduits to connect to external devices, for example external DDR SDRAM memory, and to FPGA logic defined outside of the Qsys system.

Create a Qsys System

Click **Tools** > **Qsys** in the Quartus II software to open Qsys. A **.qsys** or **.qip** file represents your Qsys system in your Quartus II software project.

Related Information

- **Creating Qsys Components** on page 6-1
- **Component Interface Tcl Reference** on page 9-1

Start a New Project or Open a Recent Project in Qsys

1. To start a new Qsys project, save the default system that appears when you open Qsys (**File** > **Save**), or click **File** > **New System**, and then save your new project.
   Qsys saves the new project in the Quartus II project directory. To alternatively save your Qsys project in a different directory, click **File** > **Save As**.
2. To open a recent Qsys project, click **File** > **Open** to browse for the project, or locate a recent project with the **File** > **Recent Projects** command.
3. To revert the project currently open in Qsys to the saved version, click the first item in the **Recent Projects** list.

Note: You can edit the directory path information in the **recent_projects.ini** file to reflect a new location for items that appear in the Recent Projects list.

Specify the Target Device

In Qsys, the **Device Family** tab allows you to select the device family and device for your Qsys system. IP components, parameters, and output options that appear with your Qsys system vary according to the device type. Qsys saves device settings in the **.qsys** file.

When you generate your Qsys system, the generated output is for the Qsys-selected device, which may be different than your Quartus II project device settings.

The Quartus II software always uses the device specified in the Quartus II project settings, even if you have already generated your Qsys system with the Qsys-selected device.

Note: Qsys generates a warning message if the Qsys device family and device do not match the Quartus II project settings. Also, when you open Qsys from within the Quartus II software, the Quartus II project device settings apply.

Add IP Components to your Qsys System

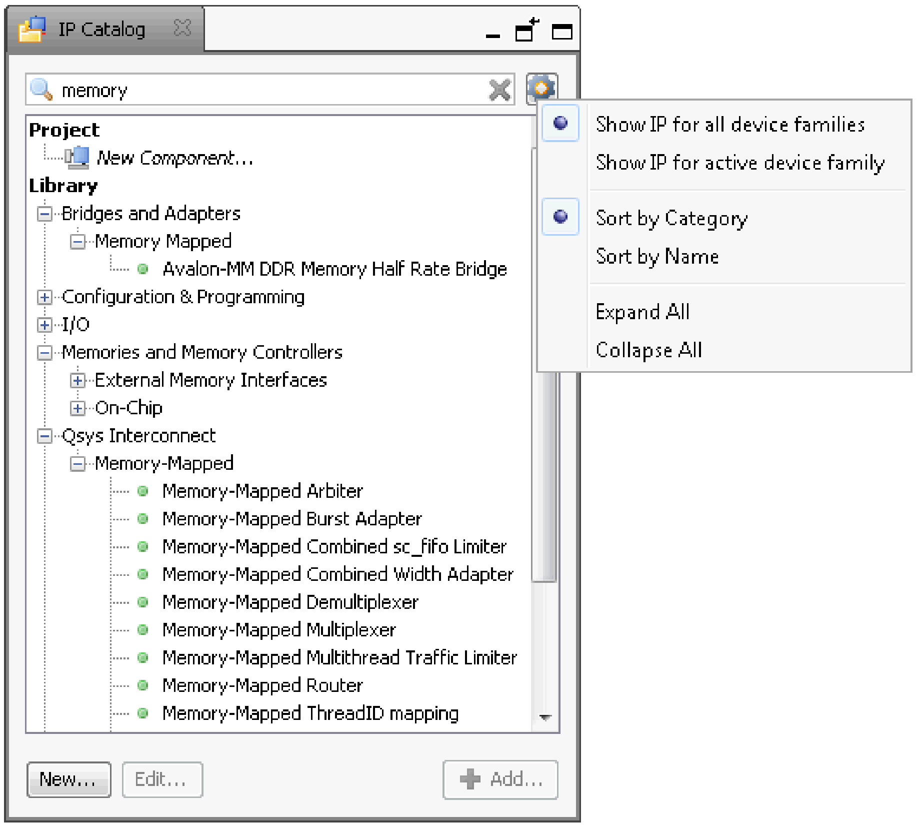
In Qsys, the IP Catalog displays IP components available for your target device. Double-click any component to launch the parameter editor. The parameter editor allows you to create a custom IP variation of the selected component. A Qsys system can contain a single instance of an IP component, or multiple, individually parameterized variations of the same IP component.

Some components have preset settings, which allow you to instantly apply preset parameter values appropriate for a specific application. When you complete your customization and click **Finish**, the component displays in the **System Contents** tab.

Right-click any IP component name in IP Catalog to display details about device support, installation location, version, and links to documentation. The IP Catalog maintains multiple versions of a component.

To locate a specific type of component, type some or all of the component's name in the IP Catalog search box. For example, you can type **memory** to locate memory-mapped IP components, or **axi** to locate AXI IP. You can also filter the IP Catalog display with options on the right-click menu.

Figure 5-1: IP Catalog


IP Catalog
memory
Project
New Component...
Library
Bridges and Adapters
Memory Mapped
Avalon-MM DDR Memory Half Rate Bridge
Configuration & Programming
I/O
Memories and Memory Controllers
External Memory Interfaces
On-Chip
Qsys Interconnect
Memory-Mapped
Memory-Mapped Arbiter
Memory-Mapped Burst Adapter
Memory-Mapped Combined sc_fifo Limiter
Memory-Mapped Combined Width Adapter
Memory-Mapped Demultiplexer
Memory-Mapped Multiplexer
Memory-Mapped Multithread Traffic Limiter
Memory-Mapped Router
Memory-Mapped ThreadID mapping
New...
Edit...
Add...
Show IP for all device families
Show IP for active device family
Sort by Category
Sort by Name
Expand All
Collapse All

Connect IP Components in Your Qsys System

The **System Contents** tab is the primary interface that you use to connect and configure components.

You connect interfaces of compatible types and opposite directions. For example, you can connect a memory-mapped master interface to a slave interface, and an interrupt sender interface to an interrupt receiver interface. You can connect any interfaces exported from a Qsys system within a parent Qsys system.

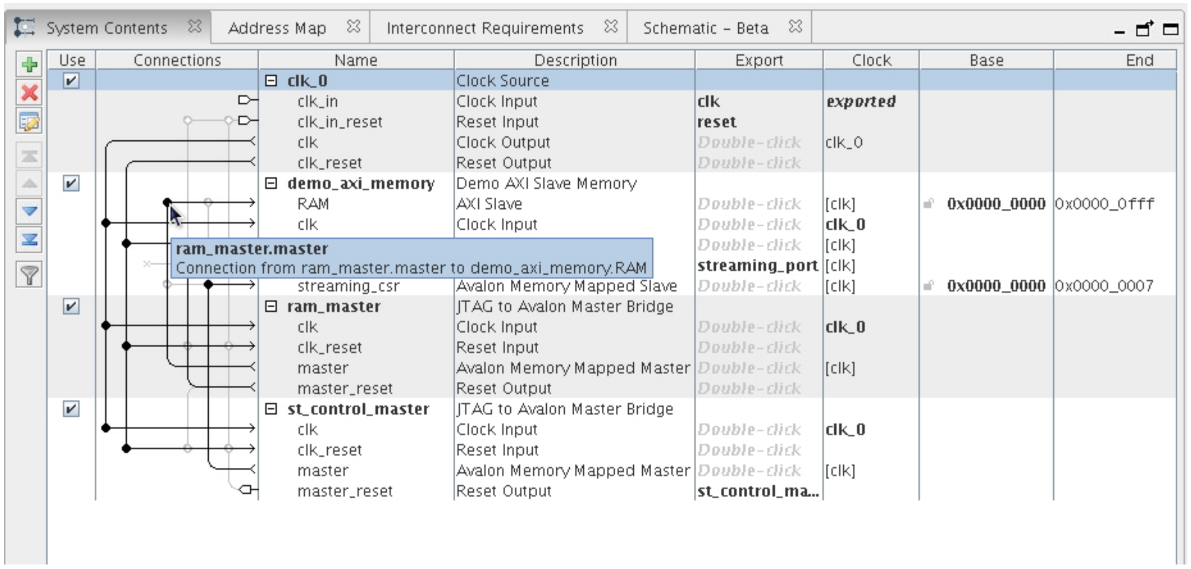
Possible connections between interfaces appear as gray lines and open circles. To make a connection, click the open circle at the intersection of the interfaces. When you make a connection, Qsys draws the connection line in black and fills the connection circle. Clicking a filled-in circle removes a connection.

Qsys interconnect connects interface signals during system generation.

The **Connections** tab (**View** > **Connections**) shows a list of current and possible connections for selected instances or interfaces in the **Hierarchy** or **System Contents** tabs. You can add and remove connections by clicking the check box for each connection. Reporting columns provide information about each connection. For example, **Clock Crossing**, **Data Width**, and **Burst** columns provide information after system generation when the Qsys interconnect adds adapters that could possible result in slower f_{Max}, or larger area.

Figure 5-2: Connections Column in the System Contents Tab

When you finish adding connections, you can deselect **Allow Connection Editing** in the right-click menu. This option sets the **Connections** column to read-only and hides the possible connections.



Use	Connections	Name	Description	Export	Clock	Base	End
✓		⊟ **clk_0**	Clock Source				
		clk_in	Clock Input	**clk**	*exported*		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click*	clk_0		
		clk_reset	Reset Output	*Double-click*			
✓		⊟ **demo_axi_memory**	Demo AXI Slave Memory				
		RAM	AXI Slave	*Double-click*	[clk]	**0x0000_0000**	0x0000_0fff
		clk	Clock Input	*Double-click*	**clk_0**		
				Double-click	[clk]		
				streaming_port	[clk]		
		streaming_csr	Avalon Memory Mapped Slave	*Double-click*	[clk]	**0x0000_0000**	0x0000_0007
✓		⊟ **ram_master**	JTAG to Avalon Master Bridge				
		clk	Clock Input	*Double-click*	**clk_0**		
		clk_reset	Reset Input	*Double-click*			
		master	Avalon Memory Mapped Master	*Double-click*	[clk]		
		master_reset	Reset Output	*Double-click*			
✓		⊟ **st_control_master**	JTAG to Avalon Master Bridge				
		clk	Clock Input	*Double-click*	**clk_0**		
		clk_reset	Reset Input	*Double-click*			
		master	Avalon Memory Mapped Master	*Double-click*	[clk]		
		master_reset	Reset Output	**st_control_ma...**			

ram_master.master
Connection from ram_master.master to demo_axi_memory.RAM

Related Information

Connecting Components

Create Connections Between Masters and Slaves

The **Address Map** tab provides the address range that each memory-mapped master must use to connect to each slave in your system.

Qsys shows the slaves on the left, the masters across the top, and the address span of the connection in each cell. If there is no connection between a master and a slave, the table cell is empty.

You can design a system where two masters access a slave at different addresses. If you use this feature, Qsys labels the **Base** and **End** address columns in the **System Contents** tab as "mixed" rather than providing the address range.

Follow these steps to change or create a connection between master and slave IP components:

1. In Qsys, click or open the **Address Map** tab.
2. Locate the table cell that represents the connection between the master and slave component pair.
3. Either type in a base address, or update the current base address in the cell.

Note: The base address of a slave component must be a multiple of the address span of the component. This restriction is a requirement of the Qsys interconnect. The result is an efficient address decoding logic, which allows Qsys to achieve the best possible f_{MAX}.

View Your Qsys System

Qsys allows you to change the display of your system to match your design development. Each tab on View menu allows you to view your design with a unique perspective. Multiple tabs open in your workspace allow you to focus on a selected element in your system under different perspectives.

Qsys GUI supports global selection and edit. When you make a selection or apply an edit in the **Hierarchy** tab, Qsys updates all other open tabs to reflect your action. For example, when you select `cpu_0` in the **Hierarchy** tab, Qsys updates the **Parameters** tab to show the parameters for `cpu_0`.

By default, when you open Qsys, the IP Catalog and **Hierarchy** tab display to the left of the main frame. The **System Contents**, **Address Map**, **Interconnect Requirements**, and **Device Family** tabs display in the main frame.

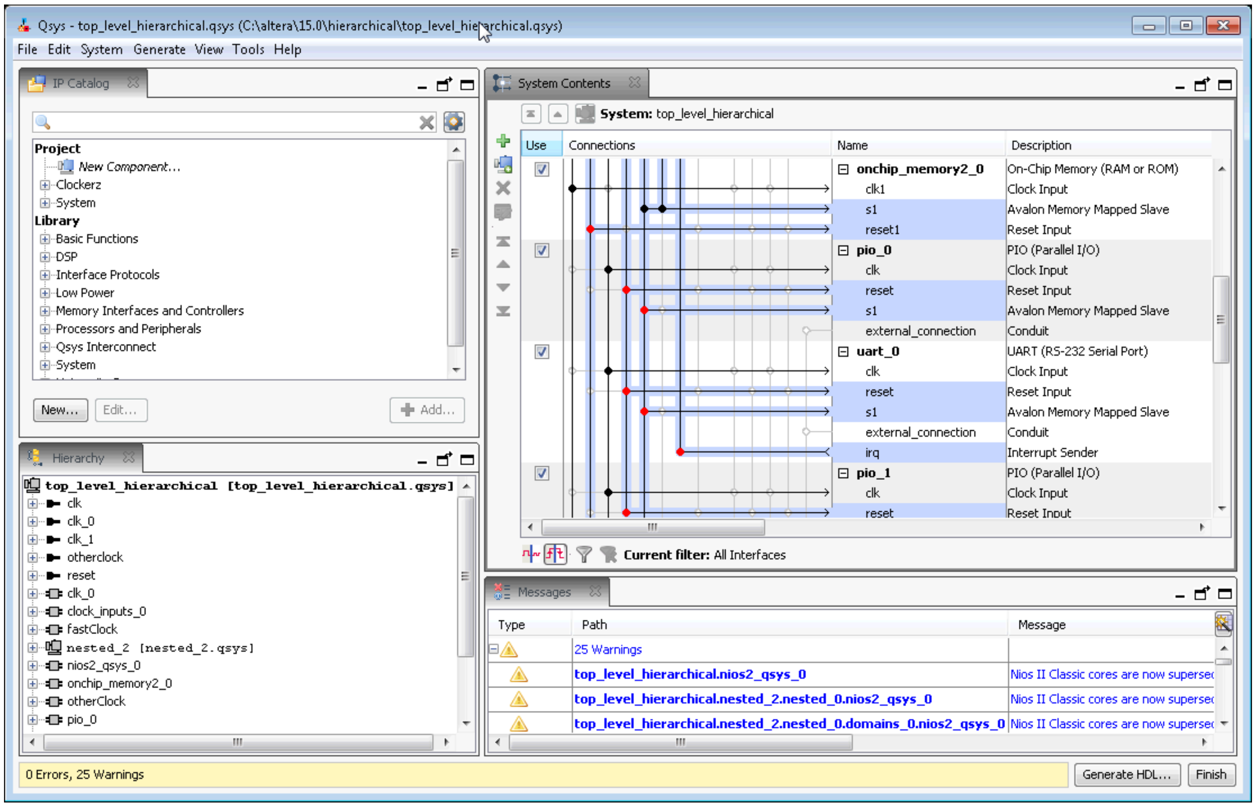
The **Messages** tab displays in the lower portion of Qsys. Double-clicking a message in the **Messages** tab changes focus to the associated element in the relevant tab to facilitate debugging. When the **Messages** tab is closed or not open in your workspace, error and warning message counts continue to display in the status bar of the Qsys window.

You can dock tabs in the main frame as a group, or individually by clicking the tab control in the upper-right corner of the main frame. You can arrange your workspace by dragging and dropping, and then grouping tabs in an order appropriate to your design development, or close or dock tabs that you are not using. Tool tips on the upper-right corner of the tab describe possible workspace arrangements, for example, restoring or disconnecting a tab to or from your workspace. When you save your system, Qsys also saves the current workspace configuration. When you re-open a saved system, Qsys restores the last saved workspace.

The **Reset to System Layout** command on the View menu restores the workspace to its default configuration for Qsys system design. The **Reset to IP Layout** command restores the workspace to its default configuration for defining and generating single IP cores.

Note: Qsys contains some tabs which are not documented and appear on the View menu as "Beta". The purpose in presenting these tabs is to allow designers to explore their usefulness in Qsys system development.

Figure 5-3: View Your Qsys System



Use	Connections	Name	Description
✓		onchip_memory2_0	On-Chip Memory (RAM or ROM)
		clk1	Clock Input
		s1	Avalon Memory Mapped Slave
		reset1	Reset Input
✓		pio_0	PIO (Parallel I/O)
		clk	Clock Input
		reset	Reset Input
		s1	Avalon Memory Mapped Slave
		external_connection	Conduit
✓		uart_0	UART (RS-232 Serial Port)
		clk	Clock Input
		reset	Reset Input
		s1	Avalon Memory Mapped Slave
		external_connection	Conduit
		irq	Interrupt Sender
✓		pio_1	PIO (Parallel I/O)
		clk	Clock Input
		reset	Reset Input

Type	Path	Message
	25 Warnings	
	top_level_hierarchical.nios2_qsys_0	Nios II Classic cores are now supersec
	top_level_hierarchical.nested_2.nested_0.nios2_qsys_0	Nios II Classic cores are now supersec
	top_level_hierarchical.nested_2.nested_0.domains_0.nios2_qsys_0	Nios II Classic cores are now supersec

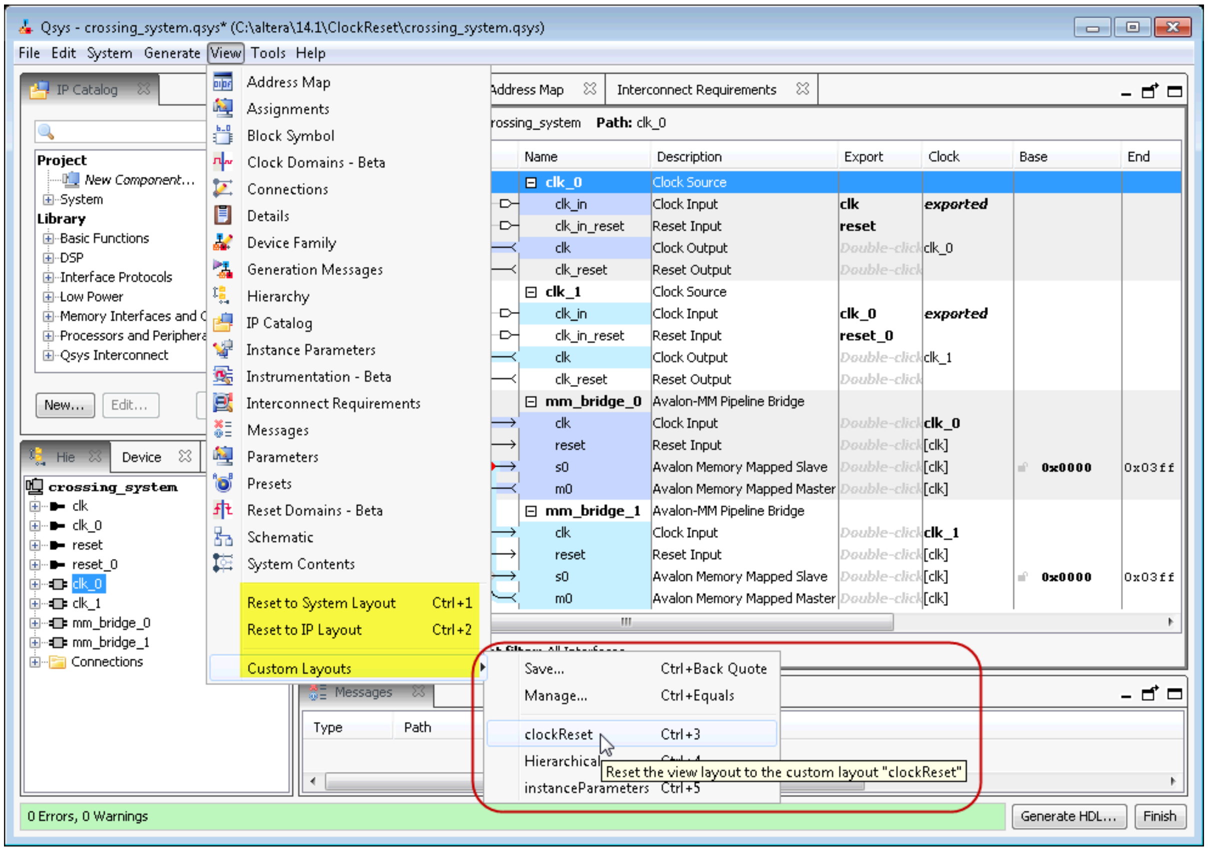
0 Errors, 25 Warnings

Manage Qsys Window Views with Layouts

Qsys Layout controls what tabs are open in your Qsys design window. When you create a Qsys window configuration that you want to keep, Qsys allows you to save that configuration as a custom layout. By default, Qsys contains a layout suitable for Qsys system design, as well as a layout that allows you to easily define and generate single IP cores for use in your Quartus II software projects.

1. To configure your Qsys window with a layout suitable for Qsys system design, click **View** > **Reset to System Layout**.
   The **System Contents**, **Address Map**, **Interconnect Requirements**, and **Messages** tabs open in the main pane, and the **IP Catalog** and **Hierarchy** tabs along the left pane.
2. To configure your Qsys window with a layout suitable for single IP core design, click **View** > **Reset to IP Layout**.
   The **Parameters** and **Messages** tabs open in the main pane, and the **Details**, **Block Symbol** and **Presets** tabs along the right pane.
3. To save your current Qsys window configuration as a custom layout, click **View** > **Custom Layouts** > **Save**.
   Qsys saves your custom layout in your project directory, and adds the layout to the custom layouts list, and the **layouts.ini** file. The **layouts.ini** file controls the order in which the layouts appear in the list.
4. To reset your Qsys window configuration to a previously saved configuration, click **View** > **Custom Layouts**, and then select the custom layout in the list.
   The Qsys windows opens with your previously saved Qsys window configuration.

Figure 5-4: Save Your Qsys Window Views and Layouts



Reset the view layout to the custom layout "clockReset"

Name	Description	Export	Clock	Base	End
clk_0	Clock Source				
clk_in	Clock Input	clk	exported		
clk_in_reset	Reset Input	reset			
clk	Clock Output	Double-click	clk_0		
clk_reset	Reset Output	Double-click			
clk_1	Clock Source				
clk_in	Clock Input	clk_0	exported		
clk_in_reset	Reset Input	reset_0			
clk	Clock Output	Double-click	clk_1		
clk_reset	Reset Output	Double-click			
mm_bridge_0	Avalon-MM Pipeline Bridge				
clk	Clock Input	Double-click	clk_0		
reset	Reset Input	Double-click	[clk]		
s0	Avalon Memory Mapped Slave	Double-click	[clk]	0x0000	0x03ff
m0	Avalon Memory Mapped Master	Double-click	[clk]		
mm_bridge_1	Avalon-MM Pipeline Bridge				
clk	Clock Input	Double-click	clk_1		
reset	Reset Input	Double-click	[clk]		
s0	Avalon Memory Mapped Slave	Double-click	[clk]	0x0000	0x03ff
m0	Avalon Memory Mapped Master	Double-click	[clk]		

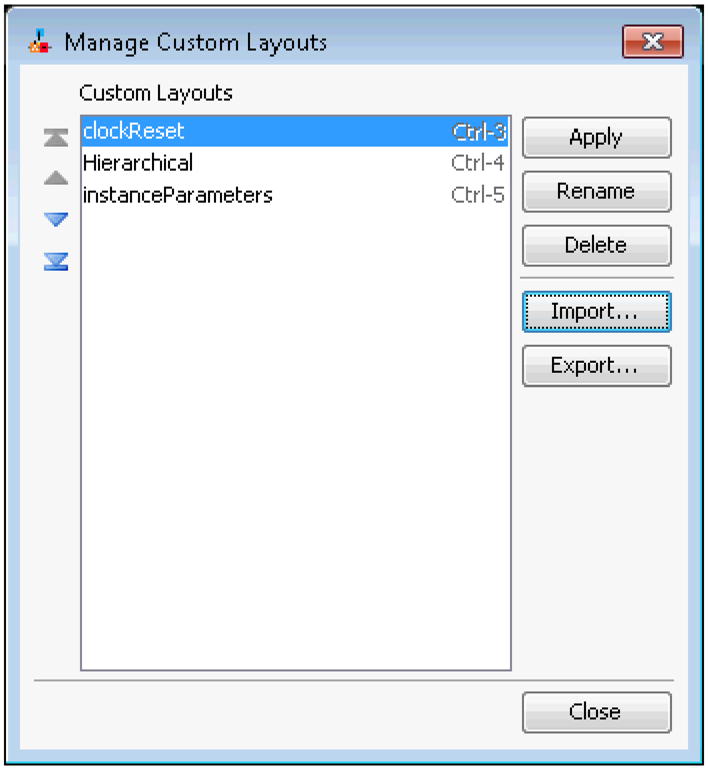
0 Errors, 0 Warnings

5. To manage your saved custom layouts, click **View** > **Custom Layouts**.

   The **Manage Custom Layouts** dialog box opens and allows you to apply a variety of functions that facilitate custom layout management, including the ability to import or export a layout from or to a different directory.

Figure 5-5: Manage Custom Layouts

The shortcut, **Ctrl-3**, for example, allows you to quickly change your Qsys window view with a quick keystroke.



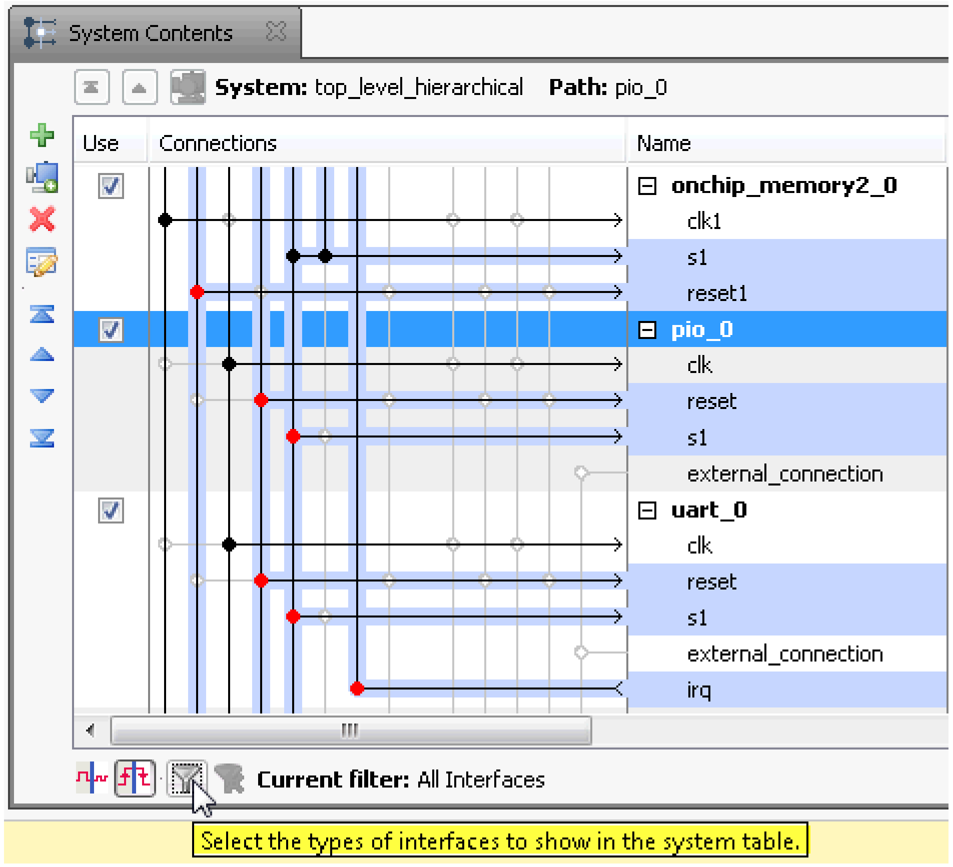
clockReset	Ctrl-3
Hierarchical	Ctrl-4
instanceParameters	Ctrl-5

Filter the Display of the System Contents tab

You can use the **Filters** dialog box in the to filter the display of your system by interface type, instance name, or by using custom tags.

For example, in the **System Contents** tab, you can show only instances that include memory-mapped interfaces, instances that are connected to a particular Nios II processor, or temporarily hide clock and reset interfaces to simplify the display.

Figure 5-6: Filter Icon in the System Contents Tab



Related Information

Filters Dialog Box

Display Details About a Component or Parameter

The **Details** tab provides information for a selected component or parameter. Qsys updates the information in the **Details** tab as you select different components.

As you click through the parameters for a component in the parameter editor, Qsys displays the description of the parameter in the **Details** tab. To return to the complete description for the component, click the header in the **Parameters** tab.

Display a Graphical Representation of a Component

In the **Block Symbol** tab, Qsys displays a graphical representation of the element that you select in the **Hierarchy** or **System Contents** tabs. You can view the selected component's port interfaces and signals. The **Show signals** option allows you to turn on or off signal graphics.

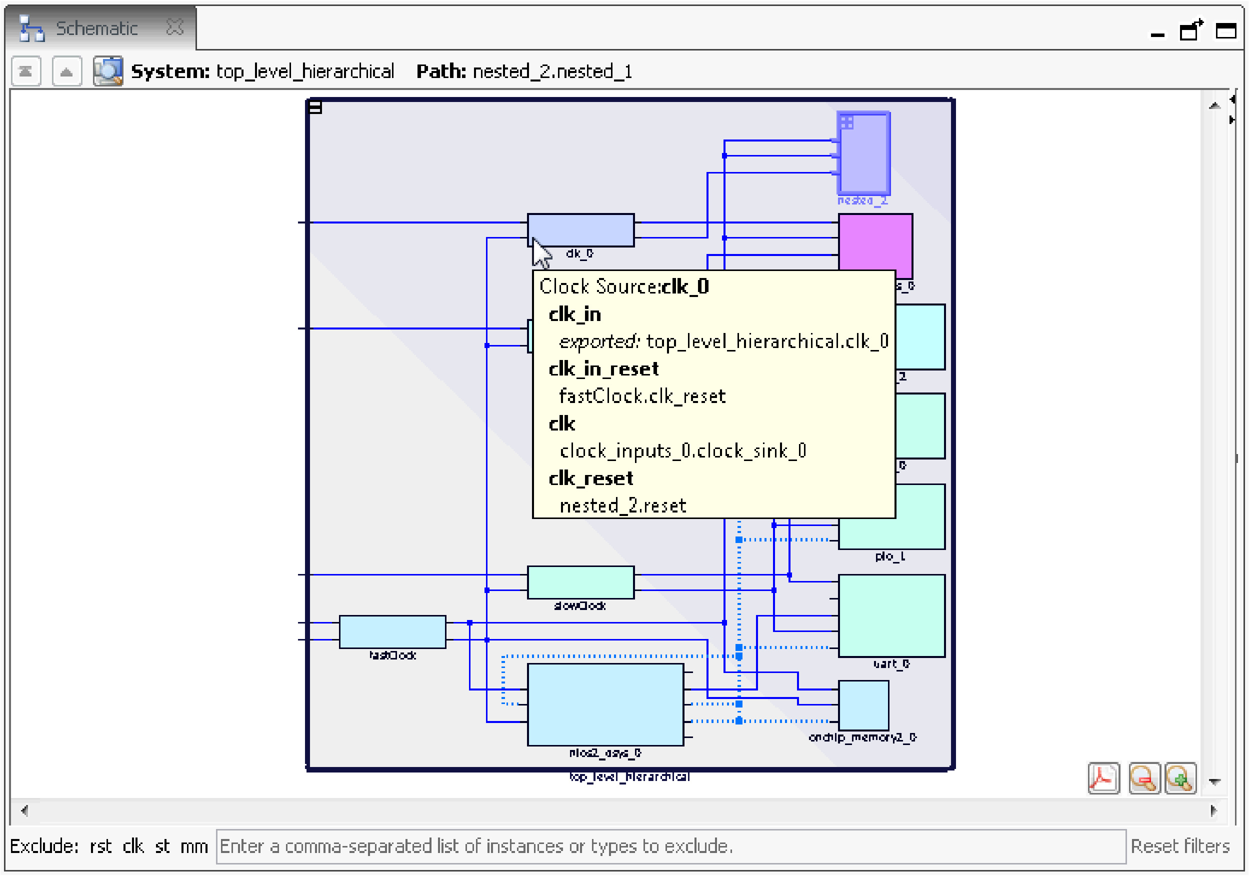
The **Block Symbol** tab appears by default in the parameter editor when you add a component to your system. When the **Block Symbol** tab is open in your workspace, it reflects changes that you make in other tabs.

View a Schematic of Your Qsys System

The **Schematic** tab displays a schematic representation of your Qsys system. Tab controls allow you to zoom into a component or connection, or to obtain tooltip details for your selection. You can use the image handles in the right panel to resize the schematic image.

If your selection is a subsystem, use the Hierarchy tool to navigate to the parent subsystem, move up one level, or to drill into the currently open subsystem.

Figure 5-7: Qsys Schematic Tab



System: top_level_hierarchical **Path:** nested_2.nested_1

Clock Source:**clk_0**
clk_in
exported: top_level_hierarchical.clk_0
clk_in_reset
fastClock.clk_reset
clk
clock_inputs_0.clock_sink_0
clk_reset
nested_2.reset

Exclude: rst clk st mm Enter a comma-separated list of instances or types to exclude.

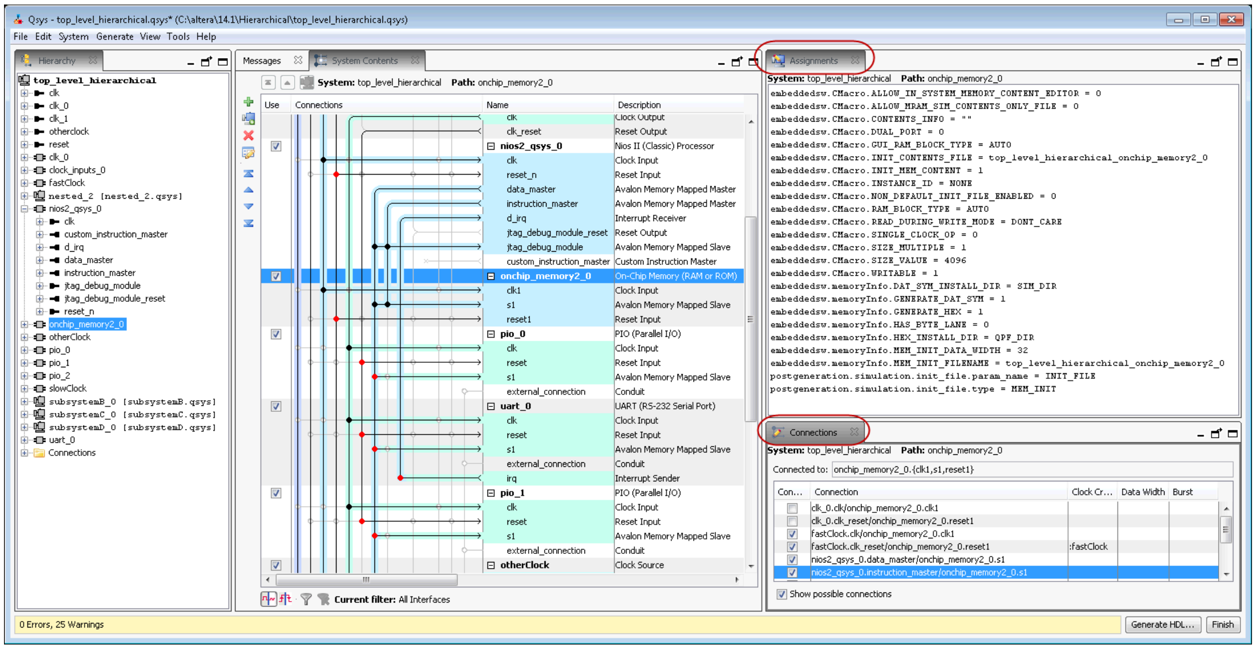
Related Information

Edit a Qsys Subsystem on page 5-32

View Assignments and Connections in Your Qsys System

On the **Assignments** tab (**View** > **Assignments**), you can view assignments for a module or element that you select in the **System Contents** tab. The **Connections** tab displays a lists of connections in your Qsys system. On the **Connections** tab (**View** > **Connections**), you can choose to connect or un-connect a module in your system, and then view the results in the **System Contents** tab.

Figure 5-8: Assignments and Connections tabs in Qsys


Qsys - top_level_hierarchical.qsys* (C:\altera\14.1\Hierarchical\top_level_hierarchical.qsys)
File Edit System Generate View Tools Help
Hierarchy
Messages
System Contents
Assignments
Connections
Use
Connections
Current filter: All Interfaces
Generate HDL...
Finish

```
top_level_hierarchical
  clk
  clk_0
  clk_1
  otherclock
  reset
  clk_0
  clock_inputs_0
  fastClock
  nested_2 [nested_2.qsys]
  nios2_qsys_0
    clk
    custom_instruction_master
    d_irq
    data_master
    instruction_master
    jtag_debug_module
    jtag_debug_module_reset
    reset_n
  onchip_memory2_0
  otherClock
  pio_0
  pio_1
  pio_2
  slowClock
  subsystemB_0 [subsystemB.qsys]
  subsystemC_0 [subsystemC.qsys]
  subsystemD_0 [subsystemD.qsys]
  uart_0
  Connections
```

System: top_level_hierarchical **Path:** onchip_memory2_0

Name	Description
clk	Clock Output
clk_reset	Reset Output
nios2_qsys_0	Nios II (Classic) Processor
clk	Clock Input
reset_n	Reset Input
data_master	Avalon Memory Mapped Master
instruction_master	Avalon Memory Mapped Master
d_irq	Interrupt Receiver
jtag_debug_module_reset	Reset Output
jtag_debug_module	Avalon Memory Mapped Slave
custom_instruction_master	Custom Instruction Master
onchip_memory2_0	On-Chip Memory (RAM or ROM)
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input
pio_0	PIO (Parallel I/O)
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
uart_0	UART (RS-232 Serial Port)
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
irq	Interrupt Sender
pio_1	PIO (Parallel I/O)
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
otherClock	Clock Source

System: top_level_hierarchical **Path:** onchip_memory2_0

```
embeddedsw.CMacro.ALLOW_IN_SYSTEM_MEMORY_CONTENT_EDITOR = 0
embeddedsw.CMacro.ALLOW_MRAM_SIM_CONTENTS_ONLY_FILE = 0
embeddedsw.CMacro.CONTENTS_INFO = ""
embeddedsw.CMacro.DUAL_PORT = 0
embeddedsw.CMacro.GUI_RAM_BLOCK_TYPE = AUTO
embeddedsw.CMacro.INIT_CONTENTS_FILE = top_level_hierarchical_onchip_memory2_0
embeddedsw.CMacro.INIT_MEM_CONTENT = 1
embeddedsw.CMacro.INSTANCE_ID = NONE
embeddedsw.CMacro.NON_DEFAULT_INIT_FILE_ENABLED = 0
embeddedsw.CMacro.RAM_BLOCK_TYPE = AUTO
embeddedsw.CMacro.READ_DURING_WRITE_MODE = DONT_CARE
embeddedsw.CMacro.SINGLE_CLOCK_OP = 0
embeddedsw.CMacro.SIZE_MULTIPLE = 1
embeddedsw.CMacro.SIZE_VALUE = 4096
embeddedsw.CMacro.WRITABLE = 1
embeddedsw.memoryInfo.DAT_SYM_INSTALL_DIR = SIM_DIR
embeddedsw.memoryInfo.GENERATE_DAT_SYM = 1
embeddedsw.memoryInfo.GENERATE_HEX = 1
embeddedsw.memoryInfo.HAS_BYTE_LANE = 0
embeddedsw.memoryInfo.HEX_INSTALL_DIR = QPF_DIR
embeddedsw.memoryInfo.MEM_INIT_DATA_WIDTH = 32
embeddedsw.memoryInfo.MEM_INIT_FILENAME = top_level_hierarchical_onchip_memory2_0
postgeneration.simulation.init_file.param_name = INIT_FILE
postgeneration.simulation.init_file.type = MEM_INIT
```

System: top_level_hierarchical **Path:** onchip_memory2_0

Connected to: onchip_memory2_0.{clk1,s1,reset1}

Con...	Connection	Clock Cr...	Data Width	Burst
☐	clk_0.clk/onchip_memory2_0.clk1			
☐	clk_0.clk_reset/onchip_memory2_0.reset1			
✓	fastClock.clk/onchip_memory2_0.clk1			
✓	fastClock.clk_reset/onchip_memory2_0.reset1	:fastClock		
✓	nios2_qsys_0.data_master/onchip_memory2_0.s1			
✓	nios2_qsys_0.instruction_master/onchip_memory2_0.s1			

✓ Show possible connections

0 Errors, 25 Warnings

Navigate Your Qsys System

The **Hierarchy** tab is a full system hierarchical navigator that expands the Qsys system contents to show all elements in your system, including, subsystems, components, interfaces, signals, and connections.

You can use the **Hierarchy** tab to browse, connect, parameterize IP, and drive changes in other open tabs. Expanding each interface in the **Hierarchy** tab allows you to view sub-components, associated elements, and signals for the interface. You can focus on a particular area of your system by coordinating selections in the **Hierarchy** tab with other open tabs in your workspace.

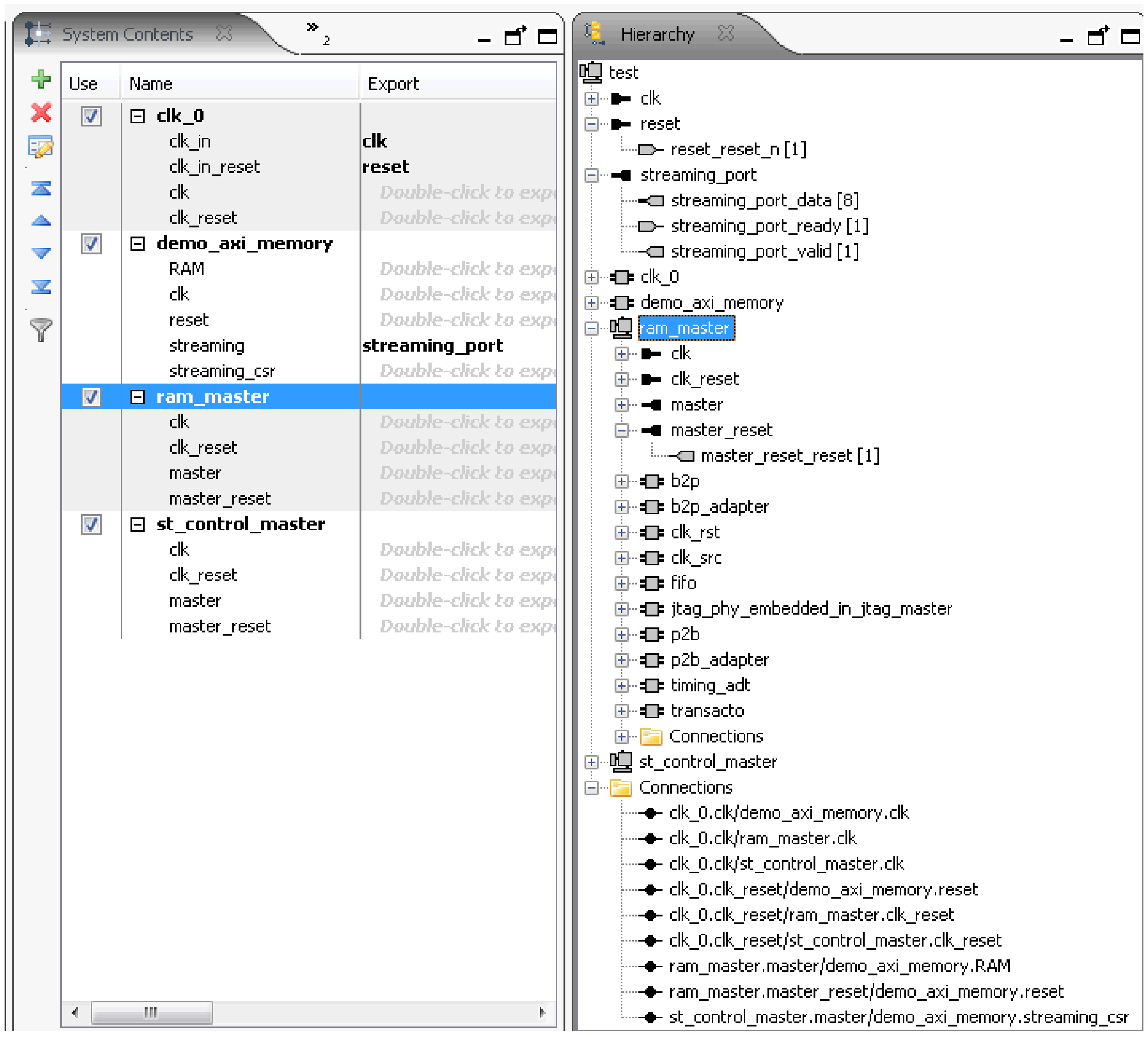
Navigating your system using the **Hierarchy** tab in conjunction with relevant tabs is useful during the debugging phase because you can contain and focus your debugging efforts to a single element in your system.

The **Hierarchy** tab provides the following information and functionality:

- Connections between signals.
- Names of signals in exported interfaces.
- Right-click menu to connect, edit, add, remove, or duplicate elements in the hierarchy.
- Internal connections of Qsys subsystems that are included as IP components. In contrast, the **System Contents** tab displays only the exported interfaces of Qsys subsystems.

Figure 5-9: Expanding System Contents in the Hierarchy Tab

The **Hierarchy** tab displays a unique icon for each element in the system. Context sensitivity between tabs facilitates design development and debugging. For example, when you select an element in the **Hierarchy** tab, Qsys selects the same element in other open tabs. This allows you to interact with your system in more detail. In the example below, the `ram_master` selection appears selected in both the **System Contents** and **Hierarchy** tabs.



Use	Name	Export
✓	⊟ **clk_0**	
	clk_in	**clk**
	clk_in_reset	**reset**
	clk	Double-click to exp
	clk_reset	Double-click to exp
✓	⊟ **demo_axi_memory**	
	RAM	Double-click to exp
	clk	Double-click to exp
	reset	Double-click to exp
	streaming	**streaming_port**
	streaming_csr	Double-click to exp
✓	⊟ **ram_master**	
	clk	Double-click to exp
	clk_reset	Double-click to exp
	master	Double-click to exp
	master_reset	Double-click to exp
✓	⊟ **st_control_master**	
	clk	Double-click to exp
	clk_reset	Double-click to exp
	master	Double-click to exp
	master_reset	Double-click to exp

```
test
 ⊞ clk
 ⊟ reset
     reset_reset_n [1]
 ⊟ streaming_port
     streaming_port_data [8]
     streaming_port_ready [1]
     streaming_port_valid [1]
 ⊞ clk_0
 ⊞ demo_axi_memory
 ⊟ ram_master
     ⊞ clk
     ⊞ clk_reset
     ⊞ master
     ⊟ master_reset
         master_reset_reset [1]
     ⊞ b2p
     ⊞ b2p_adapter
     ⊞ clk_rst
     ⊞ clk_src
     ⊞ fifo
     ⊞ jtag_phy_embedded_in_jtag_master
     ⊞ p2b
     ⊞ p2b_adapter
     ⊞ timing_adt
     ⊞ transacto
     ⊞ Connections
 ⊞ st_control_master
 ⊟ Connections
     clk_0.clk/demo_axi_memory.clk
     clk_0.clk/ram_master.clk
     clk_0.clk/st_control_master.clk
     clk_0.clk_reset/demo_axi_memory.reset
     clk_0.clk_reset/ram_master.clk_reset
     clk_0.clk_reset/st_control_master.clk_reset
     ram_master.master/demo_axi_memory.RAM
     ram_master.master_reset/demo_axi_memory.reset
     st_control_master.master/demo_axi_memory.streaming_csr
```

Related Information

Create and Manage Hierarchical Qsys Systems on page 5-30

Specify IP Component Parameters

The **Parameters** tab allows you to specify parameters that define the IP component's functionality.

When you add a component to your system, or when you double-click a component in an open tab, the parameter editor appears. Many IP components in the IP Catalog have parameters that you can configure.

If you create your own IP components, use the Hardware Component Description File (**_hw.tcl**) to specify configurable parameters.

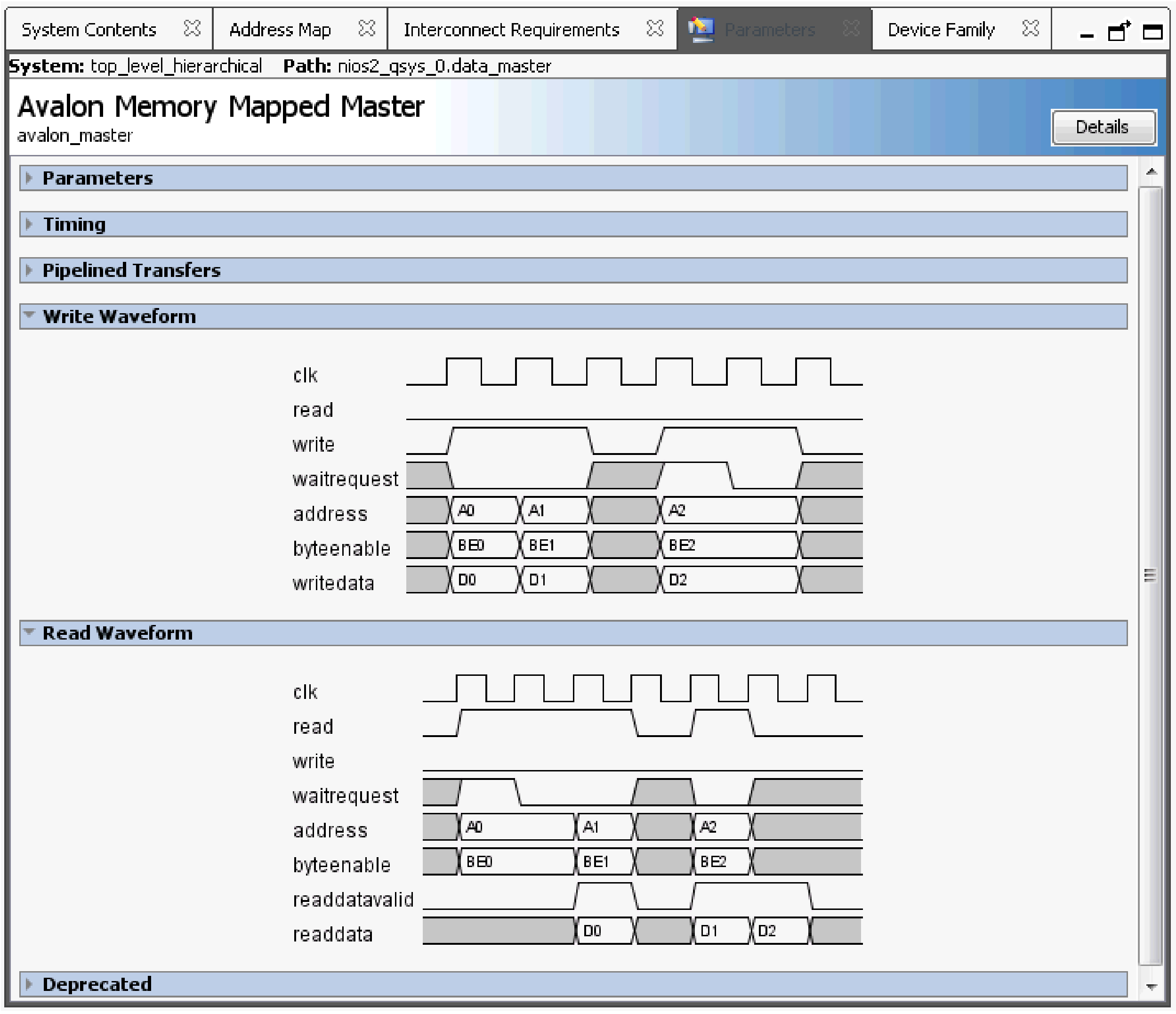
With the **Parameters** tab open, when you select an element in the **Hierarchy** tab, Qsys shows the same element in the **Parameters** tab. You can then make changes to the parameters that appear in the parameter editor, including changing the name for top-level instance that appears in the **System Contents** tab. Changes that you make in the **Parameters** tab affect your entire system and appear dynamically in other open tabs in your workspace.

In the parameter editor, the **Documentation** button provides information about a component's parameters, including the version.

At the top of the parameter editor, Qsys shows the hierarchical path for the component and its elements. This feature is useful when you navigate deep within your system with the **Hierarchy** tab.

Figure 5-10: Avalon-MM Write Master Timing Waveforms in the Parameters Tab

When you select an interface in the **Hierarchy** tab, the **Parameters** tab also allows you to review the timing for that interface. Qsys displays the read and write waveforms at the bottom of the **Parameters** tab.



System: top_level_hierarchical **Path:** nios2_qsys_0.data_master

Avalon Memory Mapped Master
avalon_master

Configure Your IP Component with a Pre-Defined Set of Parameters

The **Presets** tab allows you to apply a pre-defined set of parameter values to your IP component. The **Presets** tab opens the preset editor and allows you to create, modify, and save custom component parameter values as a preset file. Not all IP components have preset files.

When you add a new component to your system, if there are preset values available for the component, the preset editor appears in the parameter editor window and lists preset files that you can apply the component. The name of each preset file describes a particular protocol and contains the required parameter values for that protocol.

You can search for text to filter the **Presets** list. For example, if you add the **DDR3 SDRAM Controller with UniPHY** component to your system, and type `1g micron 256` in the search box, the **Presets** list shows only those parameter values that apply to the 1g micron 256 filter request. Presets whose parameter values match the current parameter settings appear in bold.

If the available preset files do not meet the requirements of your design, you can create a new preset file. In the presets editor, click **New** to open the **New Preset** dialog box. Specify the new custom preset name, component version, description of the preset, and which parameters to include in the preset. You can also specify where you want to save the preset file. If the file location that you specify is not already in the IP search path, Qsys adds the location of the new preset file to the IP search path.

In the preset editor, click **Update** to update parameter values for a custom preset. The **Update Preset** dialog box displays the default values, which you can edit, and the current value, which are static. Click Delete to remove a custom preset from the **Presets** list.

When you access the presets editor by clicking **View** > **Presets**, you can apply the available presets to the currently selected component.

Define Qsys Instance Parameters

You can use instance parameters to test the functionality of your Qsys system when you use another system as a sub-component. A higher-level Qsys system can assign values to instance parameters, and then test those values in the lower-level system.

The **Instance Script** on the **Instance Parameters** tab defines how the specified values for the instance parameters affect the sub-components in your Qsys system. The instance script allows you to create queries for the instance parameters and set the values of the parameters for the lower-level system components.

When you click **Preview Instance**, Qsys creates a preview of the current Qsys system with the specified parameters and instance script and opens the parameter editor. This command allows you to see how an instance of a system appears when you use it in another system. The preview instance does not affect your saved system.

To use instance parameters, the IP components or subsystems in your Qsys system must have parameters that can be set when they are instantiated in a higher-level system.

If you create hierarchical Qsys systems, each Qsys system in the hierarchy can include instance parameters to pass parameter values through multiple levels of hierarchy.

Related Information

Working with Instance Parameters in Qsys

Create an Instance Parameter Script in Qsys

The first command in an instance parameter script must specify the Tcl command version. The version command ensures the Tcl commands behave identically in future versions of the tool. Use the following command to specify the version of the Tcl commands, where <*version*> is the Quartus II software version number, such as 14.1:

```
package require -exact qsys <version>
```

To use Tcl commands that work with instance parameters in the instance script, you must specify the commands within a Tcl composition callback. In the instance script, you specify the name for the composition callback with the following command:

```
set_module_property COMPOSITION_CALLBACK <name of callback procedure>
```

Specify the appropriate Tcl commands inside the Tcl procedure with the following syntax:

```
proc <name of procedure defined in previous command> {}
{#Tcl commands to query and set parameters go here}
```

Example 5-1: Instance Parameter Script Example

In this example, an instance script uses the `pio_width` parameter to set the `width` parameter of a parallel I/O (PIO) component. The script combines the `get_parameter_value` and `set_instance_parameter_value` commands using brackets.

```
# Request a specific version of the scripting API
package require -exact qsys 13.1

# Set the name of the procedure to manipulate parameters:
set_module_property COMPOSITION_CALLBACK compose

proc compose {} {

# Get the pio_width parameter value from this Qsys system and
# pass the value to the width parameter of the pio_0 instance

set_instance_parameter_value pio_0 width \
[get_parameter_value pio_width]
}
```

Related Information

- Component Interface Tcl Reference on page 9-1

Supported Tcl Commands for Qsys Instance Parameter Scripts

You can use standard Tcl commands to manipulate parameters in the script, such as the `set` command to create variables, or the `expr` command for mathematical manipulation of the parameter values. Instance scripts also use Tcl commands to query the parameters of a Qsys system, or to set the values of the parameters of the sub-IP-components instantiated in the system.

get_instance_parameter_value

Description

Returns the value of a parameter in a child instance.

Usage

`get_instance_parameter_value` *<instance> <parameter>*

Returns

The value of the parameter.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter in the instance.

Example

```
get_instance_parameter_value pixel_converter input_DPI
```

get_instance_parameters

Description

Returns the names of all parameters on a child instance that can be manipulated by the parent. It omits parameters that are derived and those that have the `SYSTEM_INFO` parameter property set.

Usage

`get_instance_parameters` *<instance>*

Returns

A list of parameters in the instance.

Arguments

instance

The name of the child instance.

Example

```
get_instance_parameters instance
```

get_parameter_value

Description

Returns the current value of a parameter defined previously with the `add_parameter` command.

Usage

`get_parameter_value` <*parameter*>

Returns

The value of the parameter.

Arguments

parameter

The name of the parameter whose value is being retrieved.

Example

```
get_parameter_value fifo_width
```

get_parameters

Description

Returns the names of all the parameters in the component.

Usage

```
get_parameters
```

Returns

A list of parameter names.

Arguments

No arguments.

Example

```
get_parameters
```

send_message

Description

Sends a message to the user of the component. The message text is normally interpreted as HTML. The *<b>* element can be used to provide emphasis. If you do not want the message text to be interpreted as HTML, then pass a list like `{ Info Text }` as the message level

Usage

`send_message` *<level> <message>*

Returns

No return value.

Arguments

level

The following message levels are supported:

- `ERROR`--Provides an error message.
- `WARNING`--Provides a warning message.
- `INFO`--Provides an informational message.
- `PROGRESS`--Provides a progress message.
- `DEBUG`--Provides a debug message when debug mode is enabled.

message

The text of the message.

Example

```
send_message ERROR "The system is down!"
send_message { Info Text } "The system is up!"
```

set_instance_parameter_value

Description

Sets the value of a parameter for a child instance. Derived parameters and `SYSTEM_INFO` parameters for the child instance may not be set using this command.

Usage

`set_instance_parameter_value` *<instance> <parameter> <value>*

Returns

No return value.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter.

value

The new parameter value.

Example

```
set_instance_parameter_value uart_0 baudRate 9600
```

set_module_property

Description

Used to specify the Tcl procedure invoked to evaluate changes in Qsys system instance parameters.

Usage

`set_module_property` *<property> <value>*

Returns

No return value.

Arguments

property

The name of the property. Refer to **Module Properties**.

value

The new value of the property.

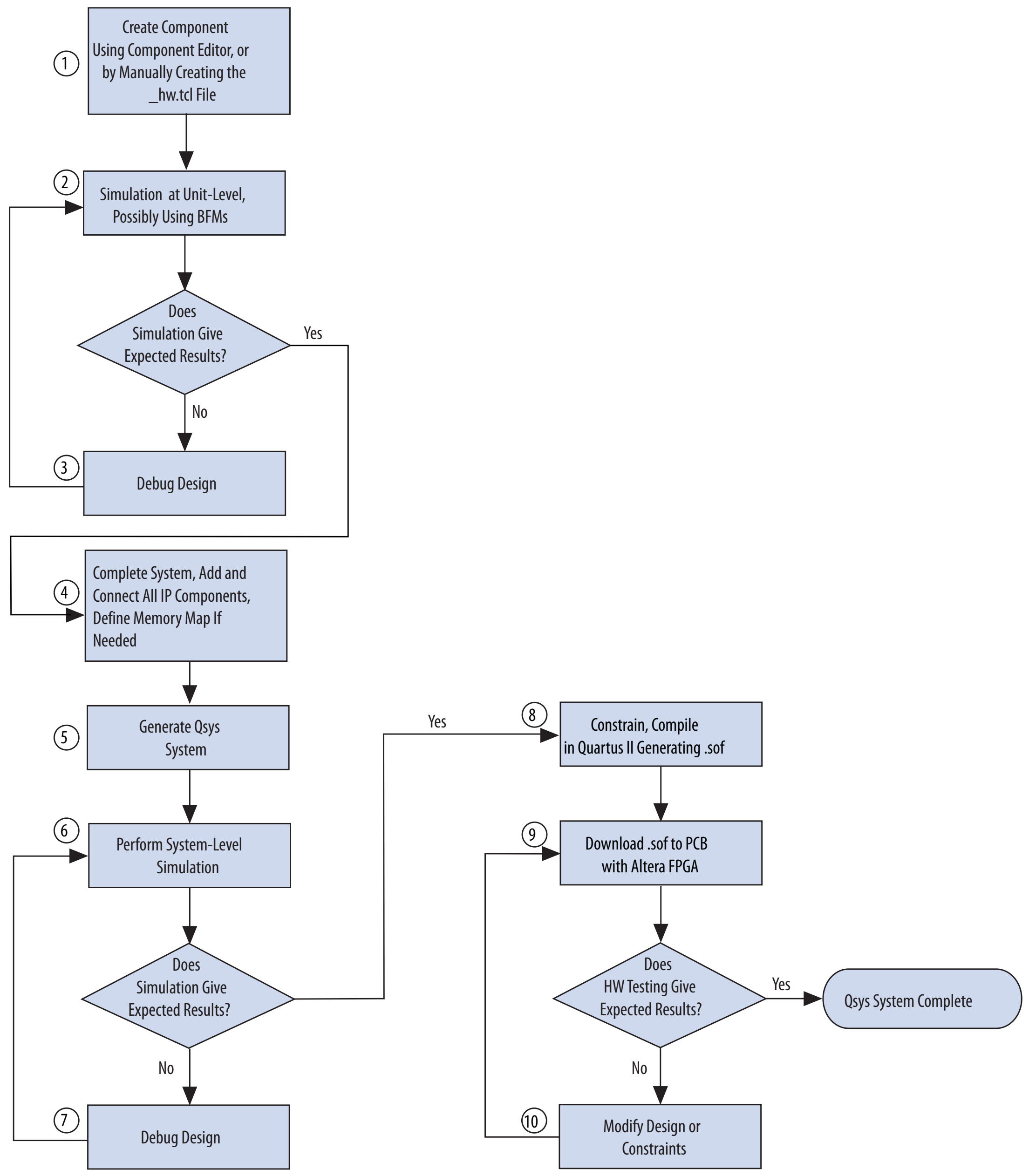
Example

```
set_module_property COMPOSITION_CALLBACK "my_composition_callback"
```

Create a Custom IP Component (_hw.tcl)

Figure 5-11: Qsys System Design Flow

The Qsys system design flow describes how to create a custom IP component using the Qsys Component Editor. You can optionally manually create a **_hw_tcl** file. The flow shows the simulation your custom IP, and at what point you can integrate it with other IP components to create a Qsys system and complete Quartus II project.


1
Create Component Using Component Editor, or by Manually Creating the _hw.tcl File
2
Simulation at Unit-Level, Possibly Using BFMs
Does Simulation Give Expected Results?
Yes
No
3
Debug Design
4
Complete System, Add and Connect All IP Components, Define Memory Map If Needed
5
Generate Qsys System
6
Perform System-Level Simulation
Does Simulation Give Expected Results?
Yes
No
7
Debug Design
8
Constrain, Compile in Quartus II Generating .sof
9
Download .sof to PCB with Altera FPGA
Does HW Testing Give Expected Results?
Yes
Qsys System Complete
No
10
Modify Design or Constraints

Note: For information on how to define and generate single IP cores for use in your Quartus II software projects, refer to *Introduction to Altera IP Cores*.

Related Information

Creating Qsys Components on page 6-1

Introduction to Altera IP Cores

Managing Quartus II Projects on page 1-1

Add IP Components to the Qsys IP Catalog

The Qsys IP Catalog lists IP components that you can use in your Qsys systems. IP components can include Altera-provided, third-party, and custom IP components that you create on your own.

Because Qsys supports hierarchical design, previously created Qsys systems also appear in the IP Catalog. Additionally, instance parameters set on a Qsys system, can be parameterized within a parent Qsys system.

Altera and third-party developers provide ready-to-use IP components, which are installed with the Quartus II software and appear in the IP Catalog. The Qsys IP Catalog includes the following IP component types:

- Microprocessors, such as the Nios® II processor
- DSP IP components, such as the Reed Solomon Decoder II
- Interface protocols, such as the IP Compiler for PCI Express
- Memory controllers, such as the RLDRAM II Controller with UniPHY
- Avalon Streaming (Avalon-ST) IP components, such as the Avalon-ST Multiplexer
- Qsys Interconnect IP components
- Verification IP (VIP) Bus Functional Models (BFMs)

You can use the **IP Search Path** option (**Tools** > **Options**) to include custom and third-party IP components in the IP Catalog. The IP Catalog displays all IP cores that are found in the IP search path.

Qsys searches the directories listed in the IP search path for the following file types:

- ***_hw.tcl**—Defines a single IP component.
- ***.ipx**—Each IP Index File (**.ipx**) indexes a collection of available IP components, or a reference to other directories to search. In general, **.ipx** files facilitate faster startup for Qsys because Qsys does not traverse directories below the **.ipx** file. Additionally, the **.ipx** file can contain information about the IP component, which eliminates the need for Qsys to read the **_hw.tcl** file to understand the basic attributes of the IP core.

Qsys recursively searches directories specified in the IP search path until it finds a **_hw.tcl** file. When Qsys finds a **_hw.tcl** file in a directory, Qsys does not search subdirectories for additional **_hw.tcl** files.

When you specify an IP search location in Qsys, a recursive descent is annotated by **. A single * signifies a match against any file within the specified directory. However, even if a recursive descent is specified, if Qsys finds a **_hw.tcl** or **.ipx** file, it does not search any subdirectories beyond that level.

Note: When you add a component to the search path, you must refresh your system by clicking **File** > **Refresh** to update the IP Catalog.

Table 5-2: IP Search Locations

Location	Description
PROJECT_DIR/*	Finds IP components and index files in the Quartus II project directory.
PROJECT_DIR/ip//***	Finds IP components and index files in any subdirectory of the **/ip** subdirectory of the Quartus project directory.

Save IP Components to the Default Search Directory

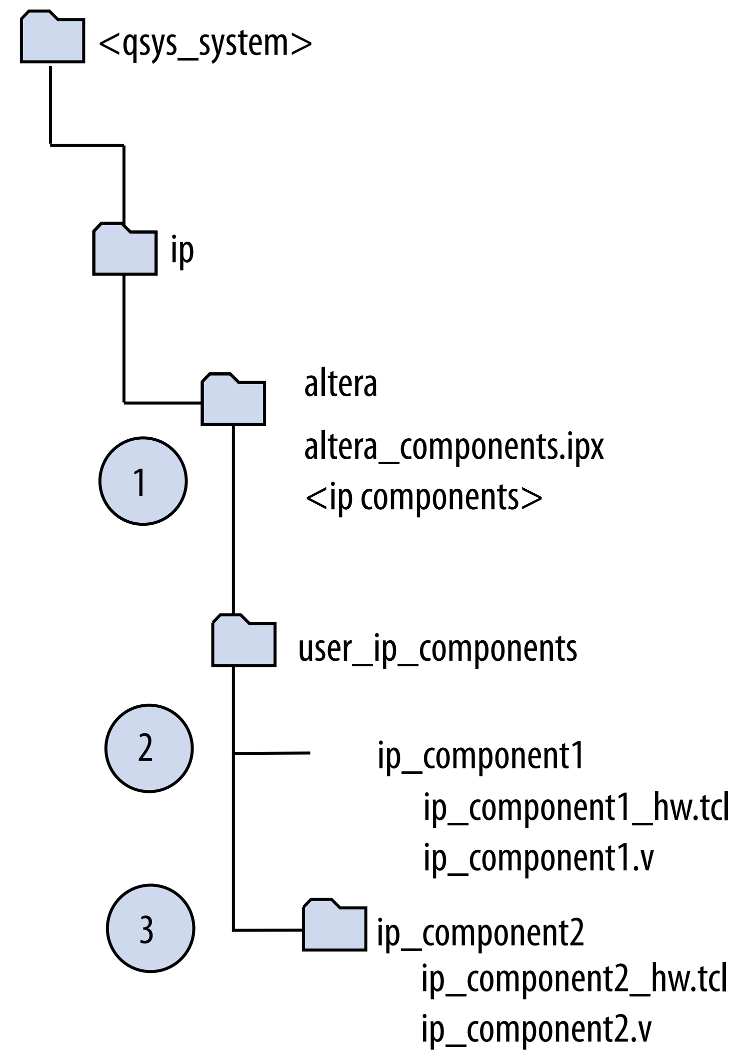
The simplest method to add an IP component to the Qsys IP Catalog is to copy the IP component files into an IP search path location.

When you create a component in the Qsys Component Editor, you can save the IP component in your Qsys project directory, or in the **/ip** subdirectory of your Qsys project directory. These methods are useful if you want to associate your IP component with a specific Qsys project.

You can also use the global IP search path in the Quartus II software (**Tools** > **Options** > **IP Catalog Search Locations**) to modify the IP search path across all Quartus II projects, Qsys systems, and Altera command line tools.

Figure 5-12: User IP Component Search

In this example, **user_ip_components** is a user-created directory for custom components.



Qsys performs the IP component search algorithm to locate **.ipx** and **_hw.tcl** files, as follows:

1. Qsys recursively searches the ***<qsys_system>*/ip** directory by default. The recursive search stops when Qsys discovers an **.ipx** file.

   Because of this traversal, if changes to the **ip_component1_hw.tcl** file are made, but the **.ipx** file is not rebuilt using **ip-make-ipx**, those changes are not reflected in Qsys because the **.ipx** file contains old information about the **ip_component1** directory.
2. If the **.ipx** file is removed, Qsys discovers both the **ip_component1** and **ip_component2** directories, which contain **_hw.tcl** files.

Note: If you save your **_hw.tcl** file in the ***<qsys_system>*/ip** directory, Qsys finds your **_hw.tcl** file and will not search subdirectories adjacent to the **_hw.tcl** file. For more information about the IP search path, refer to *Introduction to Altera IP Cores*.

Related Information

Introduction to Altera IP Cores

Set up the IP Index File (.ipx) to Search for IP Components

An IP Index File (**.ipx**) contains a search path that Qsys uses to search for IP components. You can use the `ip-make-ipx` command to create an **.ipx** file for a any directory tree, which can reduce the startup time for Qsys.

You can specify a search path in the **user_components.ipx** file in either in Qsys (**Tools** > **Options**) or the Quartus II software (**Tools** > **Options** > **IP Catalog Search Locations**). This method of discovering IP components allows you to add a locations dependent of the default search path. The **user_components.ipx** file directs Qsys to the location of an IP component or directory to search.

A <*path*> element in the **.ipx** file specifies a directory where multiple IP components may be found. A <*component*> entry specifies the path to a single component. A <*path*> element can use wildcards in its definition. An asterisk matches any file name. If you use an asterisk as a directory name, it matches any number of subdirectories.

Example 5-2: Path Element in an .ipx File

```
<library>
        <path path="…<user directory>" />
        <path path="…<user directory>" />
        …
        <component … file="…<user directory>" />
        …
</library>
```

A <*component*> element in an **.ipx** file contains several attributes to define a component. If you provide the required details for each component in an **.ipx** file, the startup time for Qsys is less than if Qsys must discover the files in a directory. The example below shows two <*component*> elements. Note that the paths for file names are specified relative to the **.ipx** file.

Example 5-3: Component Element in an .ipx File

```
<library>
  <component
    name="A Qsys Component"
    displayName="Qsys FIR Filter Component"
    version="2.1"
    file="./components/qsys_filters/fir_hw.tcl"
   />
  <component
    name="rgb2cmyk_component"
    displayName="RGB2CMYK Converter(Color Conversion Category!)"
    version="0.9"
    file="./components/qsys_converters/color/rgb2cmyk_hw.tcl"
   />
</library>
```

Note: You can verify that IP components are available with the `ip-catalog` command.

Related Information

Create an .ipx File with ip-make-ipx on page 5-74

Integrate Third-Party IP Components into the Qsys IP Catalog

You can use IP components created by Altera partners in your Qsys systems. These IP components have interfaces that are supported by Qsys, such as Avalon-MM or AXI. Additionally, some include timing and placement constraints, software drivers, simulation models, and reference designs.

To locate supported third-party IP components on Altera's web page, navigate to the *Intellectual Property & Reference Designs* page, type `Qsys Certified` in the **Search** box, select **IP Core & Reference Designs**, and then press **Enter**.

Refer to Altera's *Intellectual Property & Reference Designs* page for more information.

Related Information

Intellectual Property & Reference Designs

Upgrade Outdated IP Components

When you open a Qsys system that contains outdated IP components, Qsys automatically attempts to upgrade the IP components if it cannot locate the requested version. IP components that Qsys successfully updates appear in the **Upgrade IP Cores** dialog box with a green check mark. Most Qsys IP components support automatic upgrade.

You can include a path to older IP components in the IP Search Path, which Qsys uses even if updated versions are available. However, older versions of IP components may not work in newer version of Qsys.

Note: If your Qsys system includes an IP component(s) outside of the project directory, the directory of the **.qsys** file, or other any other directory location, you must add these dependency paths to the Qsys IP Search Path (**Tools** > **Options**).

1. With your Qsys system open, click **System** > **Upgrade IP Cores**.
   Only IP Components that are associated with the open Qsys system, and that do not support automatic upgrade appear in **Upgrade IP Cores** dialog box.
2. In the **Upgrade IP Cores** dialog box, click one or multiple IP components, and then click **Upgrade**.
   A green check mark appears for the IP components that Qsys successfully upgrades.
3. Generate your Qsys system.

Note: Qsys supports command-line upgrade for IP components with the following command:

```
qsys-generate --upgrade-ip-cores <qsys_file>
```

The <*qsys_file*> variable accepts a path to the **.qsys** file so that you are not constrained to running this command in the same directory as the **.qsys** file. Qsys reports the start and finish of the command-line upgrade, but does not name the particular IP component(s) upgraded.

For device migration information, refer to *Introduction to Altera IP Cores*.

Related Information

Add IP Components to the Qsys IP Catalog on page 5-25

Introduction to Altera IP Cores

Create and Manage Hierarchical Qsys Systems

Qsys supports hierarchical system design. You can add any Qsys system as a subsystem in another Qsys system. Qsys hierarchical system design allows you to create, explore and edit hierarchies dynamically within a single instance of the Qsys editor. Qsys generates the complete hierarchy during the top-level system's generation.

Note: You can explore parameterizable Qsys systems and **_hw.tcl** files, but you cannot edit their elements.

Your Qsys systems appear in the IP Catalog under the System category under Project. You can reuse systems across multiple designs. In a team-based hierarchical design flow, you can divide large designs into subsystems and have team members develop subsystems simultaneously.

Related Information

Navigate Your Qsys System on page 5-12

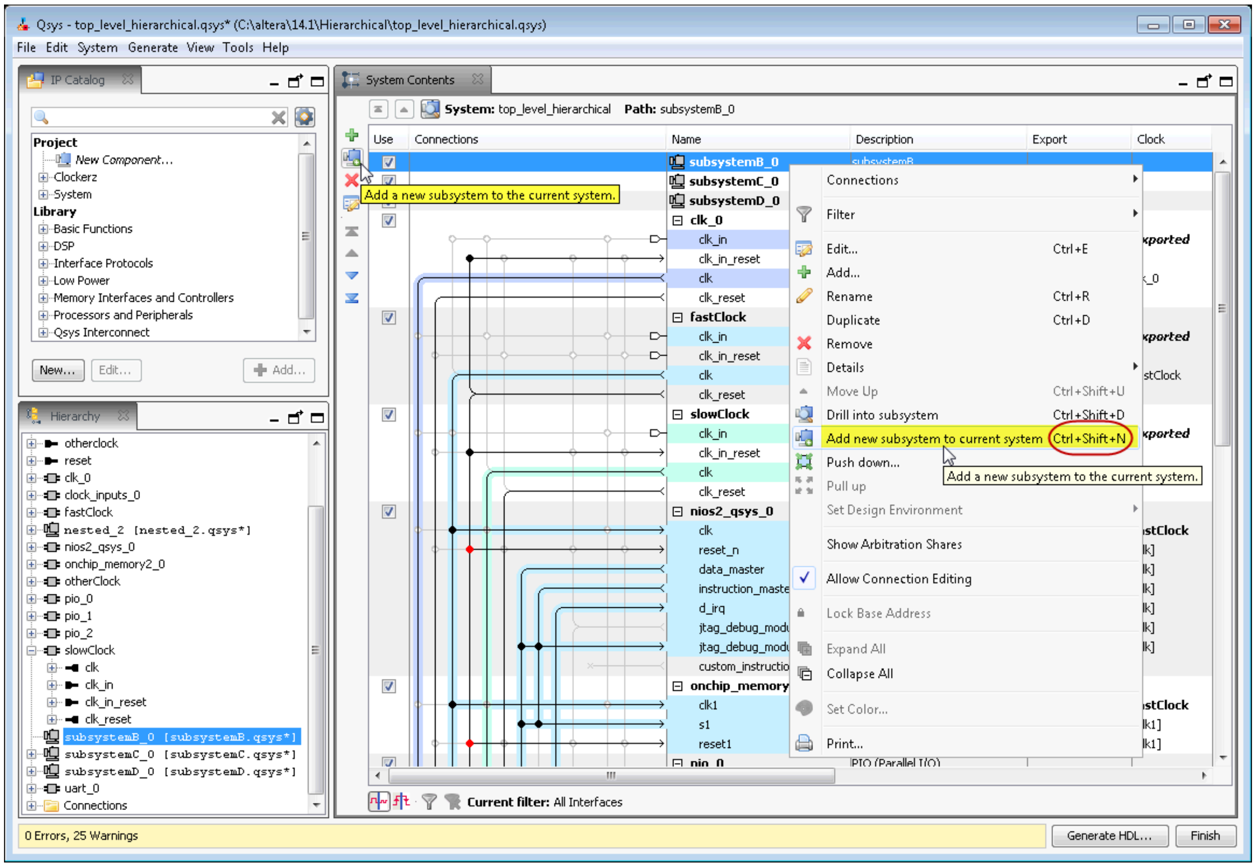
Add a Subsystem to Your Qsys Design

You can create a child subsystem or nest subsystems at any level in the hierarchy. Qsys adds a subsystem to the system you are currently editing. This can be the top-level system, or a subsystem.

To create or nest subsystems in your Qsys design, use the following methods within the **System Contents** tab:

- Right-click command: **Add a new subsystem to the current system**.
- Left panel icon.
- **CTRL+SHIFT+N**.

Figure 5-13: Add a Subsystem to Your Qsys Design


Qsys - top_level_hierarchical.qsys* (C:\altera\14.1\Hierarchical\top_level_hierarchical.qsys)
File Edit System Generate View Tools Help
IP Catalog
System Contents
System: top_level_hierarchical Path: subsystemB_0
Add a new subsystem to the current system.
Connections
Filter
Edit... Ctrl+E
Add...
Rename Ctrl+R
Duplicate Ctrl+D
Remove
Details
Move Up Ctrl+Shift+U
Drill into subsystem Ctrl+Shift+D
Add new subsystem to current system Ctrl+Shift+N
Push down...
Pull up
Set Design Environment
Show Arbitration Shares
Allow Connection Editing
Lock Base Address
Expand All
Collapse All
Set Color...
Print...
Add a new subsystem to the current system.
Hierarchy
Current filter: All Interfaces
0 Errors, 25 Warnings
Generate HDL...
Finish

Drill into a Qsys Subsystem to Explore its Contents

The ability to drill into a system provides visibility into its elements and connections. When you drill into an instance, you open the system it instantiates for editing.

You can drill into a subsystem with the following commands:

- Double-click a system in the **Hierarchy** tab.
- Right-click a system in the **Hierarchy**, **System Contents**, or **Schematic** tabs, and then select **Drill into subsystem**.
- CTRL+SHIFT+D in the **System Contents** tab.

Note: You can only drill into **.qsys** files, not parameterizable Qsys systems or **_hw.tcl** files.

The **Hierarchy** tab is rooted at the top-level and drives global selection. You can manage a hierarchical Qsys system that you build across multiple Qsys files, and view and edit their interconnected paths and address maps simultaneously. As an example, you can select a path to a subsystem in the **Hierarchy** tab, and then drill deeper into the subsystem in the **System Contents** or **Schematic** tabs. You could also select a subsystem in the **System Contents** tab, and then drill into the selected susbsystem in the **Hierarchy** tab.

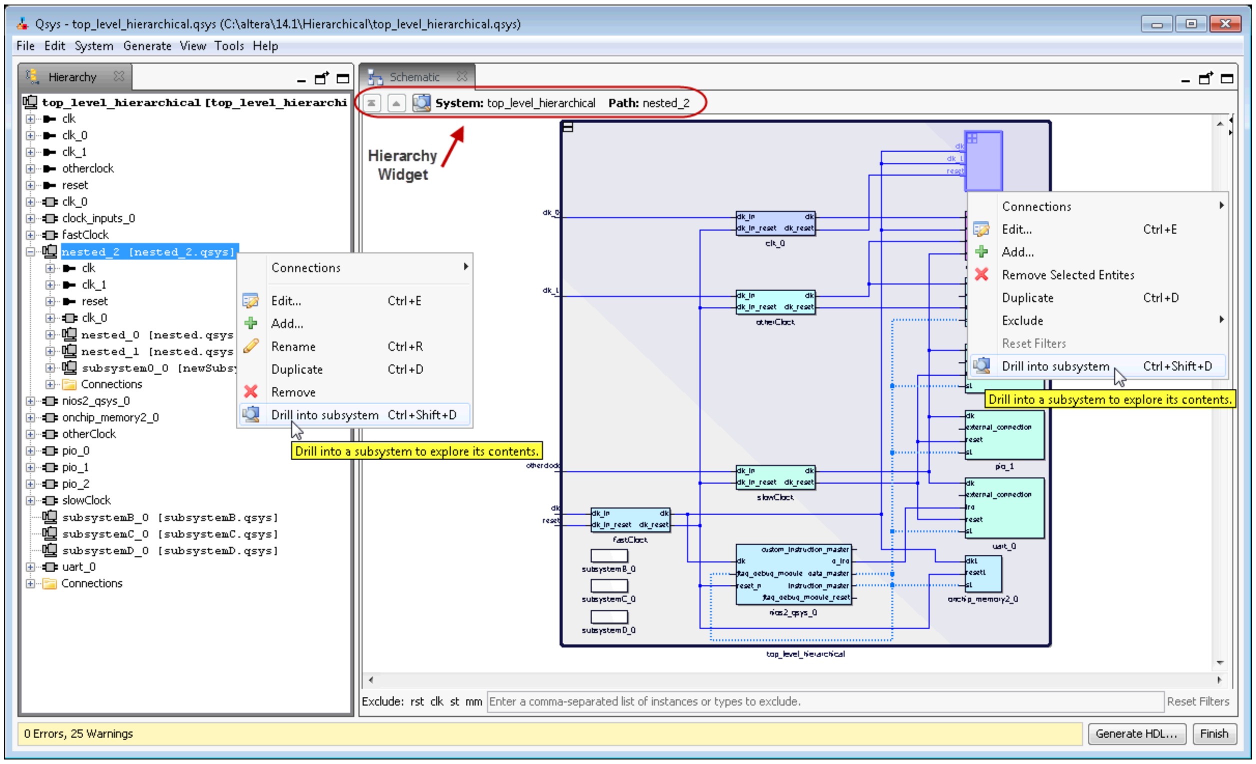
Views that manage system-level editing, for example, the **System Contents** and **Schematic** tabs, contain the hierarchy widget, which allows you to efficiently navigate your subsystems. The hierarchy widget also displays the name of the current selection, and its path in the context of the system or subsystem.

The hierarchy widget contains the following controls and information:

- **Top**—Navigates to the project-level **.qsys** file that contains the subsystem.
- **Up**—Navigates up one level from the current selection.
- **Drill Into**—Allows you to drill into an editable system.
- **System**—Displays the hierarchical location of the system you are currently editing.
- **Path**—Displays the relative path to the current selection.

Note: In the **System Contents** tab, you can use CTRL+SHIFT+U to navigate up one level, and CTRL +SHIFT+D to drill into a system.

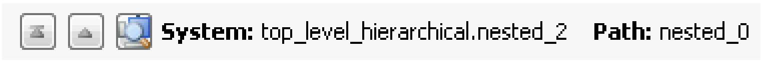
Figure 5-14: Drill into a Qsys System to Explore its Contents


Qsys - top_level_hierarchical.qsys (C:\altera\14.1\Hierarchical\top_level_hierarchical.qsys)
File Edit System Generate View Tools Help
Hierarchy
Schematic
System: top_level_hierarchical Path: nested_2
Hierarchy Widget
top_level_hierarchical [top_level_hierarchi
clk
clk_0
clk_1
otherclock
reset
clk_0
clock_inputs_0
fastClock
nested_2 [nested_2.qsys]
nested_0 [nested.qsys
nested_1 [nested.qsys
subsystem0_0 [newSubsy
Connections
nios2_qsys_0
onchip_memory2_0
otherClock
pio_0
pio_1
pio_2
slowClock
subsystemB_0 [subsystemB.qsys]
subsystemC_0 [subsystemC.qsys]
subsystemD_0 [subsystemD.qsys]
uart_0
Edit... Ctrl+E
Add...
Rename Ctrl+R
Duplicate Ctrl+D
Remove
Drill into subsystem Ctrl+Shift+D
Drill into a subsystem to explore its contents.
Remove Selected Entites
Exclude
Reset Filters
Exclude: rst clk st mm
Enter a comma-separated list of instances or types to exclude.
0 Errors, 25 Warnings
Generate HDL...
Finish

Edit a Qsys Subsystem

You can double-click a Qsys subsystem in the **Hierarchy** tab to edit its contents in any tab. When you make a change, open tabs refresh their content to reflect your edit. You can change the level of a subsystem, or push it into another subsystem with commands in the **System Contents** tab.

Note: To edit a **.qsys** file, the file must be writeable and reside outside of the ACDS installation directory. You cannot edit systems that you create from composed **_hw.tcl** files, or systems that define instance parameters.



1. In the **System Contents** or **Schematic** tabs, use the hierarchy widget to navigate to the top-level system, up one level, or down one level (drill into a system).

All tabs refresh and display the requested hierarchy level.

2. To edit a system, double-click the system in the **Hierarchy** tab. You can also drill into the system with the Hierarchy tool or right-click commands, which are available in the **Hierarchy**, **Schematic**, **System Contents** tabs.
   The system is open and available for edit in all Qsys views. A system currently open for edit appears as bold in the **Hierarchy** tab.
3. In the **System Contents** tab, you can rename any element, add, remove, or duplicate connections, and export interfaces, as appropriate.
   Changes to a subsystem affect all instances. Qsys identifies unsaved changes to a subsystem with an asterisk next to the subsystem in the **Hierarchy** tab.

Related Information

View a Schematic of Your Qsys System on page 5-11

Change the Hierarchy Level of a Qsys Component

You can push selected components down into their own subsytem, which can simplify your top-level system view. Similarly, you can pull a component up out of a subsystem to perhaps share it between two unique subsystems. Hierarchical-level management facilitates system optimization and can reduce complex connectivity in your subsystems. When you make a change, open tabs refresh their content to reflect your edit.

1. In the **System Contents** tab, to group multiple components that perhaps share a system-level component, select the components, right-click, and then select **Push down into new subsystem**.
   Qsys pushes the components into their own subsystem and re-establishes the exported signals and connectivity in the new location.
2. In the **System Contents** tab, to pull a component up out of a subsystem, select the component, and then click **Pull up**.
   Qsys pulls the component up out of the subsystem and re-establishes the exported signals and connectivity in the new location.

Save New Qsys Subsystem

When you save a subsystem to your Qsys design, Qsys confirms the new subsystem(s) in the **Confirm New System Filenames** dialog box. The **Confirm New System Filenames** dialog box appears when you save your Qsys design. Qsys uses the name that you give a subsystem as **.qsys** filename, and saves the subsystems in the project's ip directory.

1. Click **File** > **Save** to save your Qsys design.
2. In the **Confirm New System Filenames** dialog box, click **OK** to accept the subsystem file names.

   Note: If you have not yet saved your top-level system, or multiple subsystems, you can type a name, and then press **Enter**, to move to the next un-named system.
3. In the **Confirm New System Filenames** dialog box, to edit the name of a subsystem, click the subsystem, and then type the new name.
4. To cancel the save process, click **Cancel** in the **Confirm New System Filenames** dialog box.

Create an IP Component Based on a Qsys System

The **Export System as hw.tcl Component** command on the Qsys File menu allows you to save the system currently open in Qsys as an **_hw.tcl** file in project directory. The saved system displays as a new component under the **System** category in the IP Catalog.

Hierarchical System Using Instance Parameters Example

This example illustrates how you can use instance parameters to control the implementation of an on-chip memory component, `onchip_memory_0` when instantiated into a higher-level Qsys system.

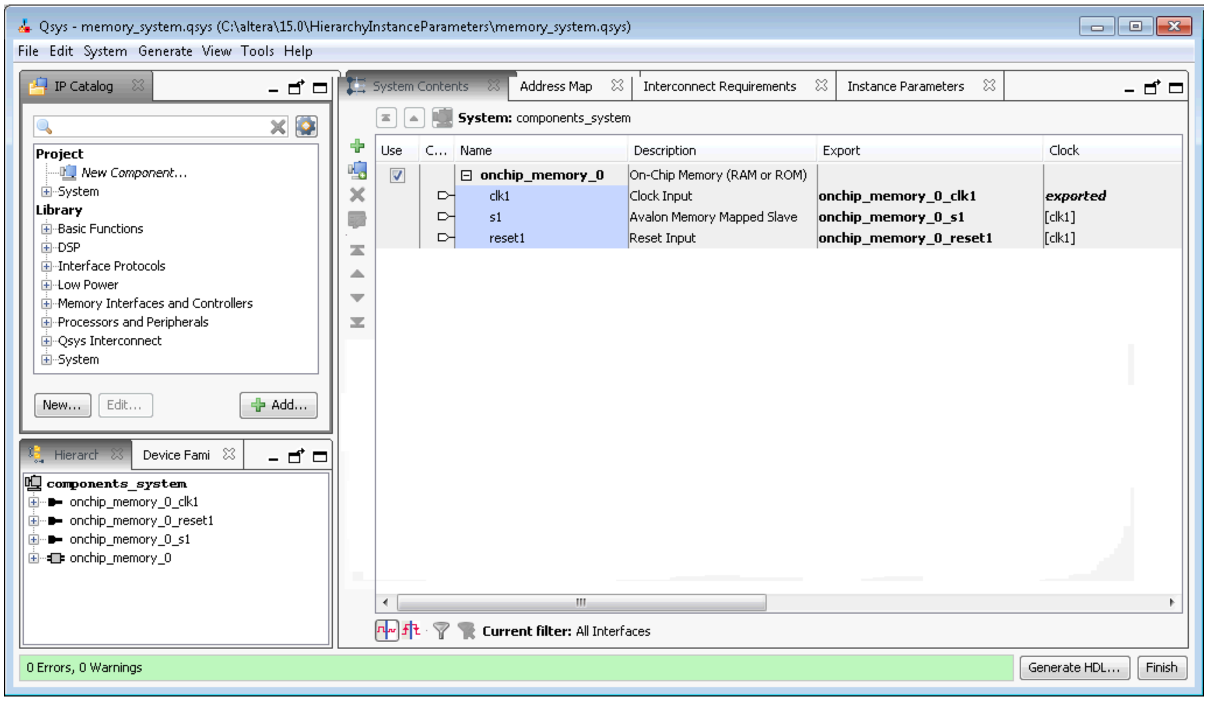
Follow the steps below to create a system that contains an on-chip memory IP component with instance parameters, and the instantiating higher-level Qsys system. With your completed system, you can vary the values of the instance parameters to review their effect within the On-Chip Memory component.

Create the Memory System

This procedure creates a Qsys system to use as subsystem as part of a hierarchical instance parameter example.

1. In Qsys, click **File** > **New System**.
2. Right-click `clk_0`, and then click **Remove**.
3. In the IP Catalog search box, type **on-chip** to locate the On-Chip Memory (RAM or ROM) component.
4. Double-click to add the On-Chip Memory component to your system.
   The parameter editor opens. When you click **Finish**, Qsys adds the component to your system with default selections.
5. Rename the On-Chip Memory component to `onchip_memory_0`.
6. In the **System Contents** tab, for the `clk1` element (`onchip_memory_0`), double-click the **Export** column.
7. In the **System Contents** tab, for the `s1` element (`onchip_memory_0`), double-click the **Export** column.
8. In the **System Contents** tab, for the `reset1` element (`onchip_memory_0`), double-click the **Export** column.
9. Click **File** > **Save** to save your Qsys system as **memory_system.qsys**.

Figure 5-15: On-Chip Memory Component System and Instance Parameters (memory_system.qsys)


Qsys - memory_system.qsys (C:\altera\15.0\HierarchyInstanceParameters\memory_system.qsys)

System: components_system

Use	C...	Name	Description	Export	Clock
✓		onchip_memory_0	On-Chip Memory (RAM or ROM)		
		clk1	Clock Input	onchip_memory_0_clk1	*exported*
		s1	Avalon Memory Mapped Slave	onchip_memory_0_s1	[clk1]
		reset1	Reset Input	onchip_memory_0_reset1	[clk1]

Current filter: All Interfaces

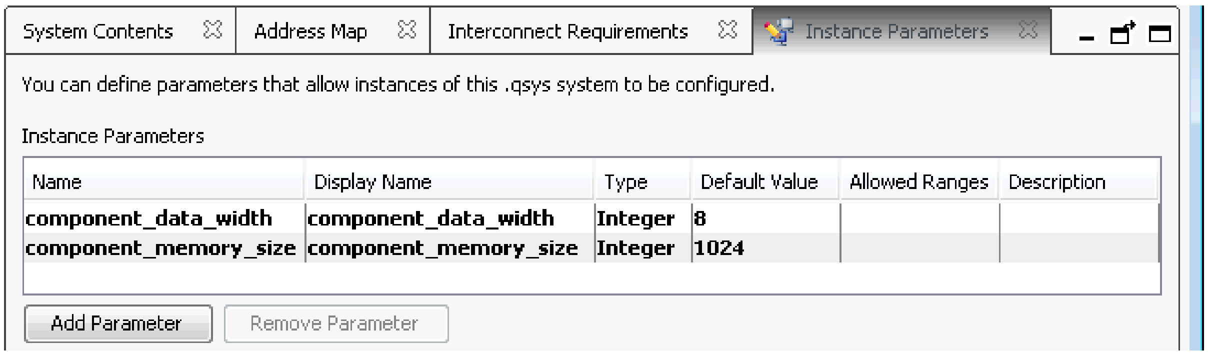
0 Errors, 0 Warnings

Add Qsys Instance Parameters

The **Instance Parameters** tab allows you to define parameters to control the implementation of a subsystem component. Each column in the **Instance Parameters** table defines a property of the parameter. This procedure creates instance parameters in a Qsys system to be used as a subsystem in a higher-level system.

1. In the **memory_system.qsys** system, click **View** > **Instance Parameters**.
2. Click **Add Parameter**.
3. In the **Name** and **Display Name** columns, rename the `new_parameter_0` parameter to `component_data_width`.
4. For `component_data_width`, select **Integer** for **Type**, and 8 as the **Default Value**.
5. Click **Add Parameter**.
6. In the **Name** and **Display Name** columns, rename the `new_parameter_0` parameter to `component_memory_size`.
7. For `component_memory_size`, select **Integer** for **Type**, and **1024** as the **Default Value**.

Figure 5-16: Qsys Instance Parameters Tab



You can define parameters that allow instances of this .qsys system to be configured.

Instance Parameters

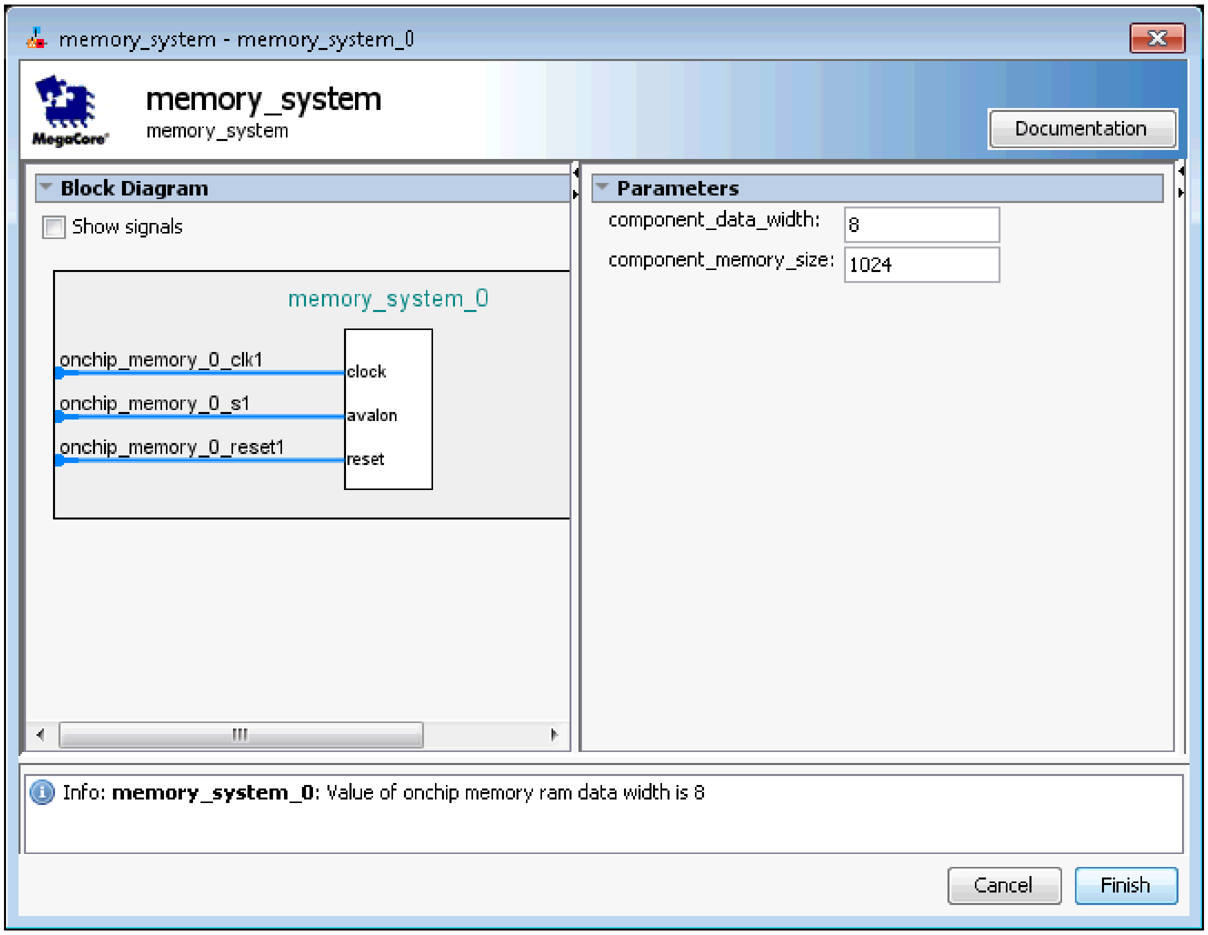
Name	Display Name	Type	Default Value	Allowed Ranges	Description
component_data_width	component_data_width	Integer	8		
component_memory_size	component_memory_size	Integer	1024		

8. In the **Instance Script** section, type the commands that control how Qsys passes parameters to an instance from the higher-level system. For example, in the script below, the `onchip_memory_0` instance receives its `dataWidth` and `memorySize` parameter values from the instance parameters that you define.

```
# request a specific version of the scripting API
package require -exact qsys 15.0

# Set the name of the procedure to manipulate parameters
set_module_property COMPOSITION_CALLBACK compose

proc compose {} {
     # manipulate parameters in here
     set_instance_parameter_value onchip_memory_0 dataWidth [get_parameter_value
component_data_width]
     set_instance_parameter_value onchip_memory_0 memorySize
[get_parameter_value component_memory_size]

     set value [get_instance_parameter_value onchip_memory_0 dataWidth]
         send_message info "Value of onchip memory ram data width is $value "
}
```

9. Click **Preview Instance** to open the parameter editor GUI.
Preview Instance allows you to see how an instance of a system appears when you use it in another system.

Figure 5-17: Preview Your Instance in the Parameter Editor



component_data_width: 8
component_memory_size: 1024

Info: **memory_system_0**: Value of onchip memory ram data width is 8

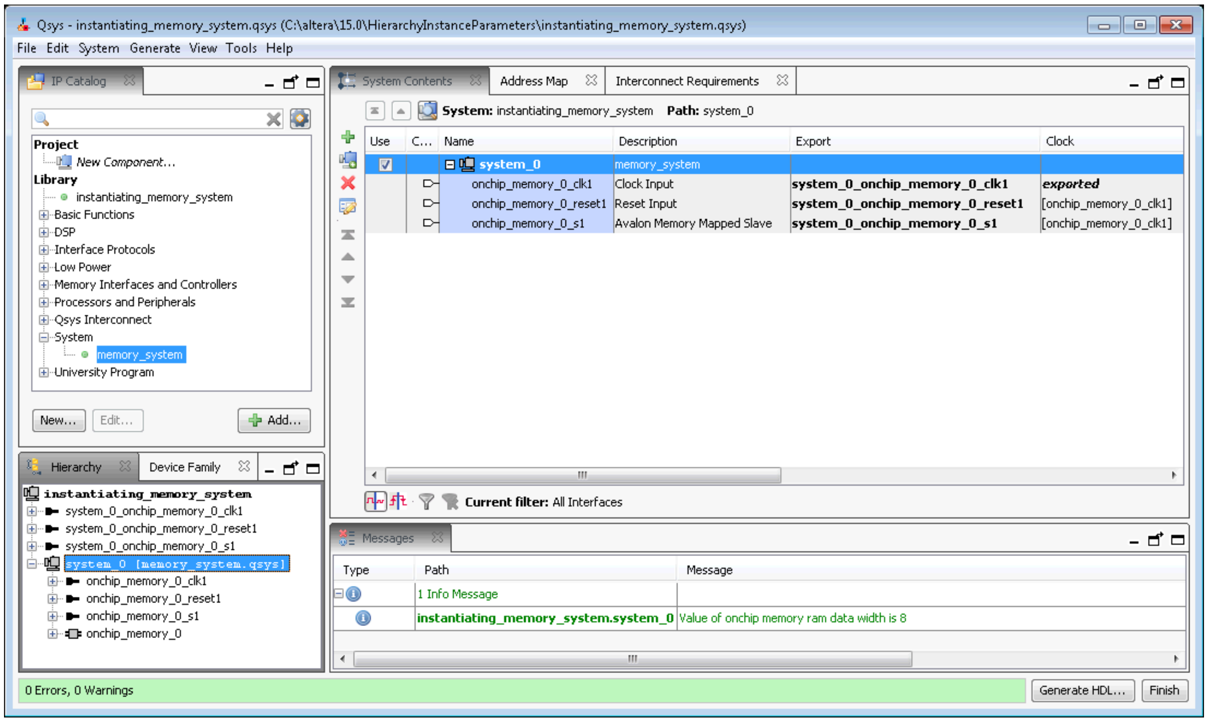
10. Click **File** > **Save**.

Create a Qsys Instantiating Memory System

This procedure creates a Qsys system to use as a higher-level system as part of a hierarchical instance parameter example.

1. In Qsys, click **File** > **New System**.
2. Right-click `clk_0`, and then click **Remove**.
3. In the IP Catalog, under **System**, double-click **memory_system**.
   The parameter editor opens. When you click **Finish**, Qsys adds the component to your system.
4. In the **Systems Contents** tab, for each element under **system_0**, double-click the **Export** column.
5. Click **File** > **Save** to save your Qsys as **instantiating_memory_system.qsys**.

Figure 5-18: Instantiating Memory System (instantiating_memory_system.qsys)


Qsys - instantiating_memory_system.qsys (C:\altera\15.0\HierarchyInstanceParameters\instantiating_memory_system.qsys)
File Edit System Generate View Tools Help
IP Catalog
Project
New Component...
Library
instantiating_memory_system
Basic Functions
DSP
Interface Protocols
Low Power
Memory Interfaces and Controllers
Processors and Peripherals
Qsys Interconnect
System
memory_system
University Program
New... Edit... Add...
Hierarchy
Device Family
System Contents
Address Map
Interconnect Requirements
Current filter: All Interfaces
Messages
Generate HDL... Finish

instantiating_memory_system
- system_0_onchip_memory_0_clk1
- system_0_onchip_memory_0_reset1
- system_0_onchip_memory_0_s1
- system_0 [memory_system.qsys]
  - onchip_memory_0_clk1
  - onchip_memory_0_reset1
  - onchip_memory_0_s1
  - onchip_memory_0

System: instantiating_memory_system **Path:** system_0

Use	C...	Name	Description	Export	Clock
✓		system_0	memory_system		
		onchip_memory_0_clk1	Clock Input	system_0_onchip_memory_0_clk1	exported
		onchip_memory_0_reset1	Reset Input	system_0_onchip_memory_0_reset1	[onchip_memory_0_clk1]
		onchip_memory_0_s1	Avalon Memory Mapped Slave	system_0_onchip_memory_0_s1	[onchip_memory_0_clk1]

Type	Path	Message
	1 Info Message	
	instantiating_memory_system.system_0	Value of onchip memory ram data width is 8

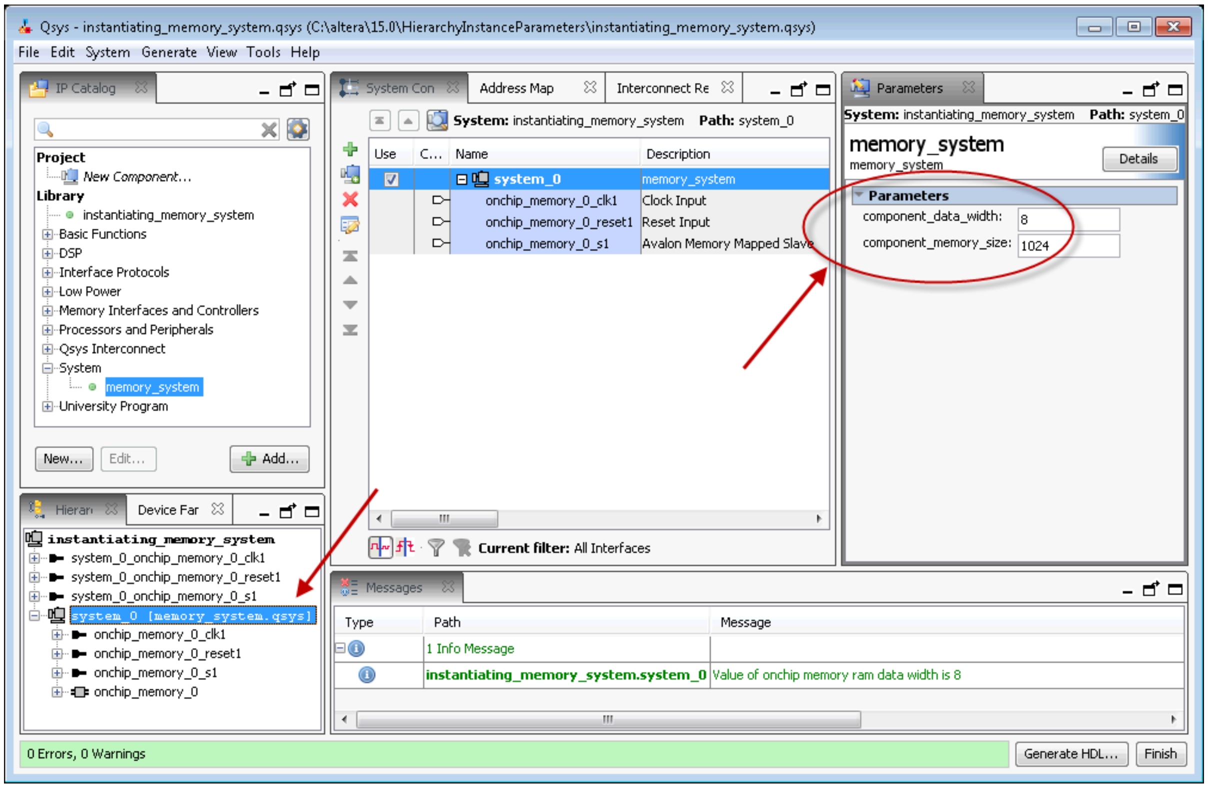
0 Errors, 0 Warnings

Apply Instance Parameters at a Higher-Level Qsys System and Pass the Parameters to the Instantiated Lower-Level System

This procedure shows you how to use Qsys instance parameters to control the implementation of an on-chip memory component as part of a hierarchical instance parameter example.

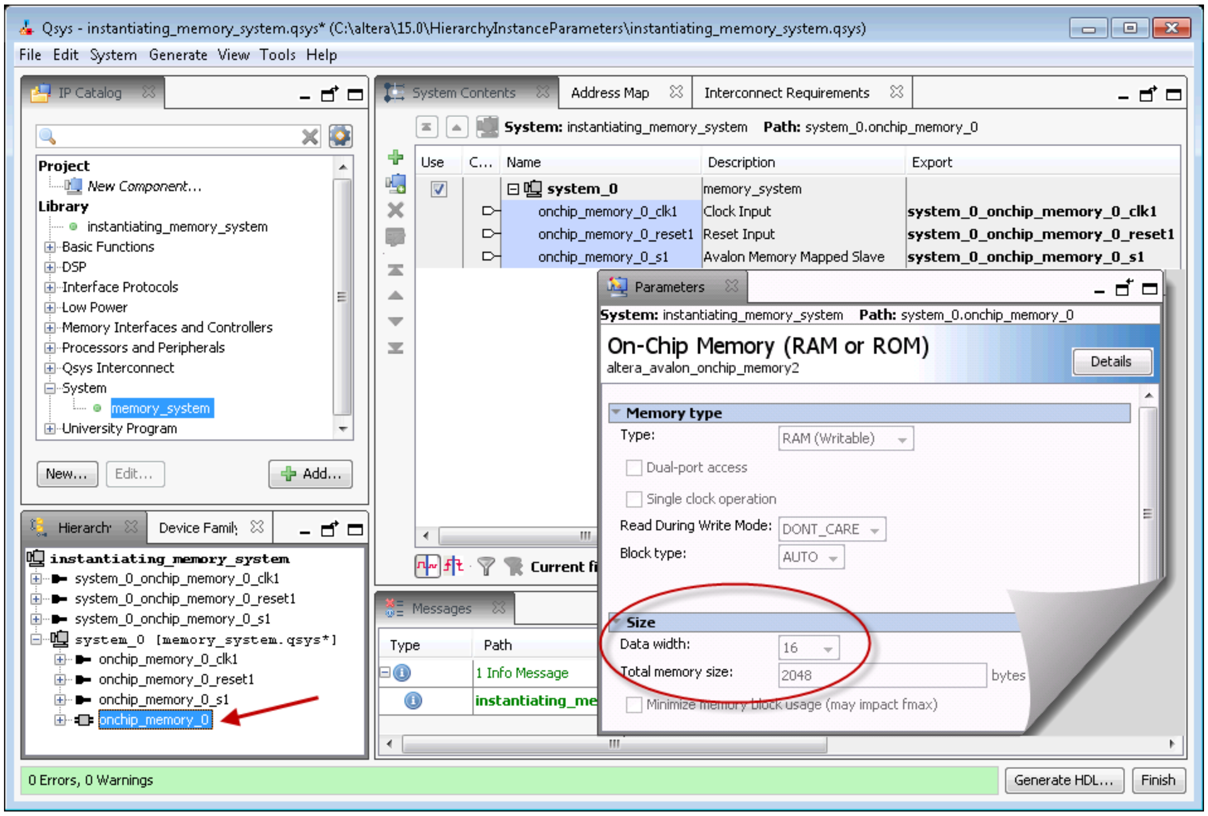
1. In the **instantiating_memory_system.qsys** system, in the **Hierarchy** tab, click and expand **system_0 (memory_system.qsys)**.
2. Click **View** > **Parameters**.
   The instance paramters for the **memory_system.qsys** display in the parameter editor.

Figure 5-19: Displays memory_system.qsys Instance Parameters in the Parameter Editor


Qsys - instantiating_memory_system.qsys (C:\altera\15.0\HierarchyInstanceParameters\instantiating_memory_system.qsys)
File Edit System Generate View Tools Help
IP Catalog
Project
New Component...
Library
instantiating_memory_system
Basic Functions
DSP
Interface Protocols
Low Power
Memory Interfaces and Controllers
Processors and Peripherals
Qsys Interconnect
System
memory_system
University Program
New... Edit... Add...
System Con
Address Map
Interconnect Re
System: instantiating_memory_system Path: system_0
Use C... Name Description
system_0 memory_system
onchip_memory_0_clk1 Clock Input
onchip_memory_0_reset1 Reset Input
onchip_memory_0_s1 Avalon Memory Mapped Slave
Current filter: All Interfaces
Parameters
System: instantiating_memory_system Path: system_0
memory_system
memory_system
Details
Parameters
component_data_width: 8
component_memory_size: 1024
Hierarc
Device Far
instantiating_memory_system
system_0_onchip_memory_0_clk1
system_0_onchip_memory_0_reset1
system_0_onchip_memory_0_s1
system_0 [memory_system.qsys]
onchip_memory_0_clk1
onchip_memory_0_reset1
onchip_memory_0_s1
onchip_memory_0
Messages
Type Path Message
1 Info Message
instantiating_memory_system.system_0 Value of onchip memory ram data width is 8
0 Errors, 0 Warnings
Generate HDL... Finish

3. On the **Parameters** tab, change the value of **memory_data_width** to 16, and **memory_memory_size** to 2048.
4. In the **Hierarchy** tab, under **system_0 (memory_system.qsys)**, click `onchip_memory_0`.
   When you select `onchip_memory_0`, the new parameter values for **Data width** and **Total memory size** size are displayed.

Figure 5-20: Changing the Values of Your Instance Parameters


Qsys - instantiating_memory_system.qsys* (C:\altera\15.0\HierarchyInstanceParameters\instantiating_memory_system.qsys)
File Edit System Generate View Tools Help
IP Catalog
Project
New Component...
Library
instantiating_memory_system
Basic Functions
DSP
Interface Protocols
Low Power
Memory Interfaces and Controllers
Processors and Peripherals
Qsys Interconnect
System
memory_system
University Program
New... Edit... Add...
Hierarchy
Device Family
System Contents
Address Map
Interconnect Requirements
Current fi
Messages
Parameters
Details
Generate HDL... Finish

System: instantiating_memory_system **Path:** system_0.onchip_memory_0

Use	C...	Name	Description	Export
✓		system_0	memory_system	
		onchip_memory_0_clk1	Clock Input	system_0_onchip_memory_0_clk1
		onchip_memory_0_reset1	Reset Input	system_0_onchip_memory_0_reset1
		onchip_memory_0_s1	Avalon Memory Mapped Slave	system_0_onchip_memory_0_s1

```
instantiating_memory_system
  system_0_onchip_memory_0_clk1
  system_0_onchip_memory_0_reset1
  system_0_onchip_memory_0_s1
  system_0 [memory_system.qsys*]
    onchip_memory_0_clk1
    onchip_memory_0_reset1
    onchip_memory_0_s1
    onchip_memory_0
```

Type	Path
	1 Info Message
	instantiating_me

System: instantiating_memory_system **Path:** system_0.onchip_memory_0

On-Chip Memory (RAM or ROM)
altera_avalon_onchip_memory2

Memory type

Type: RAM (Writable)

Dual-port access

Single clock operation

Read During Write Mode: DONT_CARE

Block type: AUTO

Size

Data width: 16

Total memory size: 2048 bytes

Minimize memory block usage (may impact fmax)

0 Errors, 0 Warnings

View and Filter Clock and Reset Domains in Your Qsys System

The Qsys clock and reset domains tabs allow you to see clock domains and reset domains in your Qsys system. Qsys determines clock and reset domains by the associated clocks and resets, which are displayed in tooltips for each interface in your system. You can filter your system to display particular components or interfaces within a selected clock or reset domain. The clock and reset domain tabs also provide quick access to performance bottlenecks by indicating connection points where Qsys automatically inserts clock crossing adapters and reset synchronizers during system generation. With these tools, you can more easily create optimal connections between interfaces.

Click **View** > **Clock Domains**, or **View** > **Reset Domains** to open the respective tabs in your workspace. The domain tools display as a tree with the current system at the root. You can select each clock or reset domain in the list to view associated interfaces.

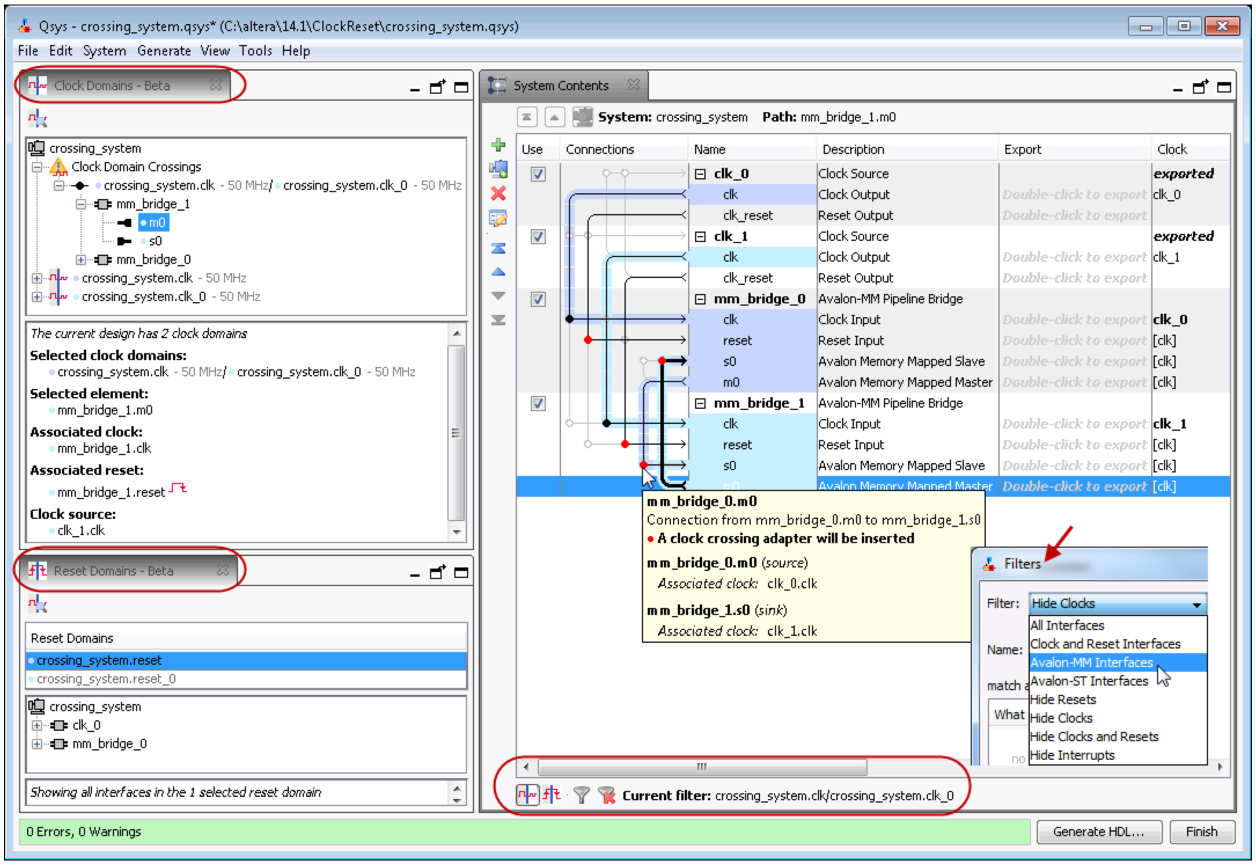
When you select an element in the **Clock Domains** tab, the corresponding selection appears in the **System Contents** tab. You can select single or multiple interface(s) and module(s). Mouse over tooltips in the **System Contents** tab to provide detailed information for all elements and connections. Colors that appear for the clocks and resets in the domain tools correspond to the colors in the **System Contents** and **Schematic** tabs.

Clock and reset control tools at the bottom on the **System Contents** tab allow you to toggle between highlighting clock or reset domains. You can further filter your view with options in the **Filters** dialog box, which is accessible by clicking the filter icon at the bottom of the **System Contents** tab. In the **Filters** dialog box, you can choose to view a single interface, or to hide clock, reset, or interrupt interfaces.

Clock and reset domain tools respond to global selection and edits, and help to provide answers to the following system design questions:

- How many clock and reset domains do you have in your Qsys system?
- What interfaces and modules does each clock or reset domain contain?
- Where do clock or reset crossings occur?
- At what connection points does Qsys automatically insert clock or reset adapters?
- Where do you have to manually insert a clock or reset adapter?

Figure 5-21: Qsys Clock and Reset Domains



crossing_system
Clock Domain Crossings
crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz
mm_bridge_1
m0
s0
mm_bridge_0
crossing_system.clk - 50 MHz
crossing_system.clk_0 - 50 MHz

The current design has 2 clock domains
Selected clock domains: crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz
Selected element: mm_bridge_1.m0
Associated clock: mm_bridge_1.clk
Associated reset: mm_bridge_1.reset
Clock source: clk_1.clk

Reset Domains
crossing_system.reset
crossing_system.reset_0

crossing_system
clk_0
mm_bridge_0

Showing all interfaces in the 1 selected reset domain

System: crossing_system **Path:** mm_bridge_1.m0

Use	Name	Description	Export	Clock
✓	**clk_0**	Clock Source		***exported***
	clk	Clock Output	*Double-click to export*	clk_0
	clk_reset	Reset Output	*Double-click to export*	
✓	**clk_1**	Clock Source		***exported***
	clk	Clock Output	*Double-click to export*	clk_1
	clk_reset	Reset Output	*Double-click to export*	
✓	**mm_bridge_0**	Avalon-MM Pipeline Bridge		
	clk	Clock Input	*Double-click to export*	**clk_0**
	reset	Reset Input	*Double-click to export*	[clk]
	s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]
✓	**mm_bridge_1**	Avalon-MM Pipeline Bridge		
	clk	Clock Input	*Double-click to export*	**clk_1**
	reset	Reset Input	*Double-click to export*	[clk]
	s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]

mm_bridge_0.m0
Connection from mm_bridge_0.m0 to mm_bridge_1.s0
● **A clock crossing adapter will be inserted**
mm_bridge_0.m0 *(source)*
Associated clock: clk_0.clk
mm_bridge_1.s0 *(sink)*
Associated clock: clk_1.clk

Current filter: crossing_system.clk/crossing_system.clk_0

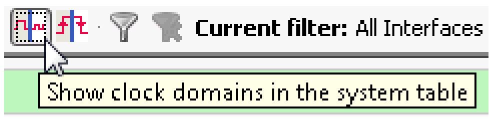
0 Errors, 0 Warnings

View Clock Domains in Your Qsys System

With the **Clock Domains** tab, you can filter the **System Contents** tab to display a single clock domain, or multiple clock domains. You can further filter your view with selections in the **Filters** dialog box. When you select an element in the **Clock Domains** tab, the corresponding selection appears highlighted in the **System Contents** tab.

1. To view clock domain interfaces and their connections in your Qsys system, click **View** > **Clock Domains** to open the Clock Domains tab.
2. To enables and disable highlighting of the clock domains in the **System Contents** tab, click the clock control tool at the bottom of the **System Contents** tab.

Figure 5-22: Clock Control Tool

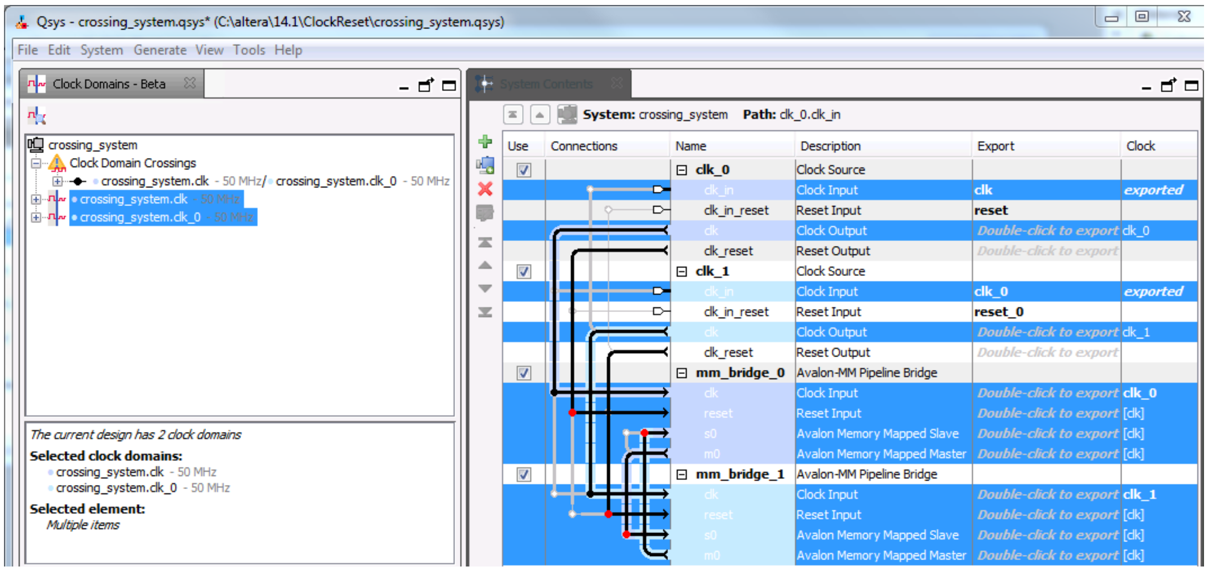


Current filter: All Interfaces

Show clock domains in the system table

3. To view a single clock domain, or multiple clock domains and their modules and connections, click the clock name(s) in the **Clock Domains** tab.
   The modules for the selected clock domain(s) and their connections appear highlighted in the **System Contents** tab. Detailed information for the current selection appears in the clock domain details pane. Red dots in the **Connections** column indicate auto insertions by Qsys during system generation, for example, a reset synchronizer or clock crossing adapter.

Figure 5-23: Clock Domains



Qsys - crossing_system.qsys* (C:\altera\14.1\ClockReset\crossing_system.qsys)

crossing_system
Clock Domain Crossings
crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz
crossing_system.clk - 50 MHz
crossing_system.clk_0 - 50 MHz

The current design has 2 clock domains
Selected clock domains:
crossing_system.clk - 50 MHz
crossing_system.clk_0 - 50 MHz
Selected element:
Multiple items

System: crossing_system **Path:** clk_0.clk_in

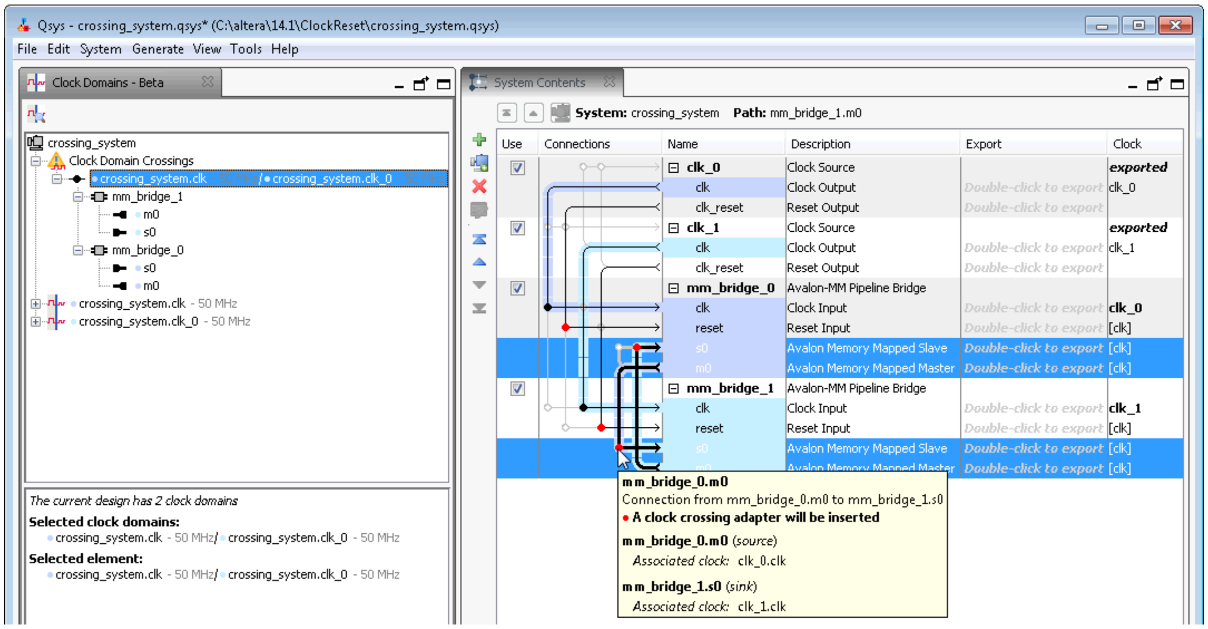
Use	Connections	Name	Description	Export	Clock
✓		**clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	*exported*
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	*Double-click to export*	clk_0
		clk_reset	Reset Output	*Double-click to export*	
✓		**clk_1**	Clock Source		
		clk_in	Clock Input	**clk_0**	*exported*
		clk_in_reset	Reset Input	**reset_0**	
		clk	Clock Output	*Double-click to export*	clk_1
		clk_reset	Reset Output	*Double-click to export*	
✓		**mm_bridge_0**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	*Double-click to export*	**clk_0**
	●	reset	Reset Input	*Double-click to export*	[clk]
	●	s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]
✓		**mm_bridge_1**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	*Double-click to export*	**clk_1**
	●	reset	Reset Input	*Double-click to export*	[clk]
	●	s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]

4. To view interfaces that cross clock domains, expand the **Clock Domain Crossings** icon in the **Clock Domains** tab, and select each element to view its details in the **System Contents** tab.

   Qsys lists the interfaces that cross clock domain under **Clock Domain Crossings**. As you click through the elements, detailed information appears in the clock domain details pane. Qsys also highlights the selection in the **System Contents** tab.

   If a connection crosses a clock domain, the connection circle appears as a red dot in the **System Contents** tab. Mouse over tooltips at the red dot connections provide details about the connection, as well as what adapter type Qsys automatically inserts during system generation.

Figure 5-24: Clock Domain Crossings


Qsys - crossing_system.qsys* (C:\altera\14.1\ClockReset\crossing_system.qsys)
File Edit System Generate View Tools Help
Clock Domains - Beta
System Contents

crossing_system
Clock Domain Crossings
crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz
mm_bridge_1
m0
s0
mm_bridge_0
s0
m0
crossing_system.clk - 50 MHz
crossing_system.clk_0 - 50 MHz

The current design has 2 clock domains

Selected clock domains:
crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz

Selected element:
crossing_system.clk - 50 MHz/ crossing_system.clk_0 - 50 MHz

System: crossing_system **Path:** mm_bridge_1.m0

Use	Connections	Name	Description	Export	Clock
✓		⊟ **clk_0**	Clock Source		***exported***
		clk	Clock Output	*Double-click to export*	clk_0
		clk_reset	Reset Output	*Double-click to export*	
✓		⊟ **clk_1**	Clock Source		***exported***
		clk	Clock Output	*Double-click to export*	clk_1
		clk_reset	Reset Output	*Double-click to export*	
✓		⊟ **mm_bridge_0**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]
✓		⊟ **mm_bridge_1**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	*Double-click to export*	**clk_1**
		reset	Reset Input	*Double-click to export*	[clk]
		s0	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		m0	Avalon Memory Mapped Master	*Double-click to export*	[clk]

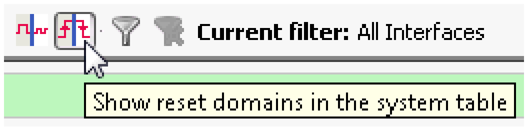
mm_bridge_0.m0
Connection from mm_bridge_0.m0 to mm_bridge_1.s0
● **A clock crossing adapter will be inserted**
mm_bridge_0.m0 (*source*)
Associated clock: clk_0.clk
mm_bridge_1.s0 (*sink*)
Associated clock: clk_1.clk

View Reset Domains in Your Qsys System

With the **Reset Domains** tab, you can filter the **System Contents** tab to display a single reset domain, or multiple reset domains. When you select an element in the **Reset Domains** tab, the corresponding selection appears in the **System Contents** tab.

1. To view reset domain interfaces and their connections in your Qsys system, click **View** > **Reset Domains** to open the **Reset Domains** tab.
2. To show reset domains in the **System Contents** tab, click the reset control tool at the bottom of the **System Contents** tab.

Figure 5-25: Reset Control Tool


Current filter: All Interfaces
Show reset domains in the system table

3. To view a single reset domain, or multiple reset domains and their modules and connections, click the reset name(s) in the **Reset Domain** tab.

   Qsys displays your selection according to the following rules:

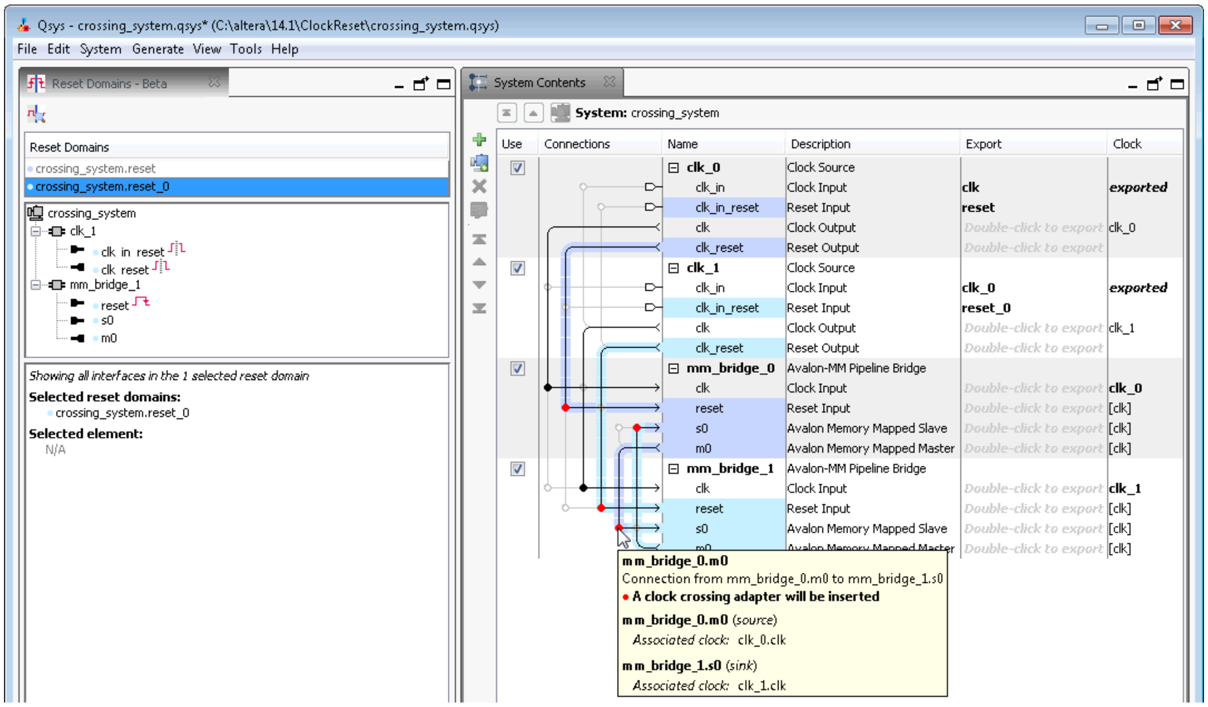
- When you select multiple reset domains, the **System Contents** tab shows interfaces and modules in both reset domains.
- When you select a single reset domain, the other reset domain(s) are grayed out, unless the two domains have interfaces in common.
- Reset interfaces appear black when connected to multiple reset domains.
- Reset interfaces appear gray when they are not connected to all of the selected reset domains.
- If an interface is contained in multiple reset domains, the interface is grayed out.

Detailed information for your selection appears in the reset domain details pane.

Note: Red dots in the **Connections** column between reset sinks and sources indicate auto insertions by Qsys during system generation, for example, a reset synchronizer. Qsys decides when to display a red dot with the following protocol, and ends the decision process at first match.

- Multiple resets fan into a common sink.
- Reset inputs are associated with different clock domains.
- Reset inputs have different synchronicity.

Figure 5-26: Reset Domains


Qsys - crossing_system.qsys* (C:\altera\14.1\ClockReset\crossing_system.qsys)
File Edit System Generate View Tools Help
Reset Domains - Beta
System Contents
System: crossing_system

Reset Domains
crossing_system.reset
crossing_system.reset_0

crossing_system
clk_1
clk in reset
clk reset
mm_bridge_1
reset
s0
m0

Showing all interfaces in the 1 selected reset domain
Selected reset domains:
crossing_system.reset_0
Selected element:
N/A

Use	Connections	Name	Description	Export	Clock
✓		**clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	*exported*
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	Double-click to export	clk_0
		clk_reset	Reset Output	Double-click to export	
✓		**clk_1**	Clock Source		
		clk_in	Clock Input	**clk_0**	*exported*
		clk_in_reset	Reset Input	**reset_0**	
		clk	Clock Output	Double-click to export	clk_1
		clk_reset	Reset Output	Double-click to export	
✓		**mm_bridge_0**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	Double-click to export	**clk_0**
		reset	Reset Input	Double-click to export	[clk]
		s0	Avalon Memory Mapped Slave	Double-click to export	[clk]
		m0	Avalon Memory Mapped Master	Double-click to export	[clk]
✓		**mm_bridge_1**	Avalon-MM Pipeline Bridge		
		clk	Clock Input	Double-click to export	**clk_1**
		reset	Reset Input	Double-click to export	[clk]
		s0	Avalon Memory Mapped Slave	Double-click to export	[clk]
		m0	Avalon Memory Mapped Master	Double-click to export	[clk]

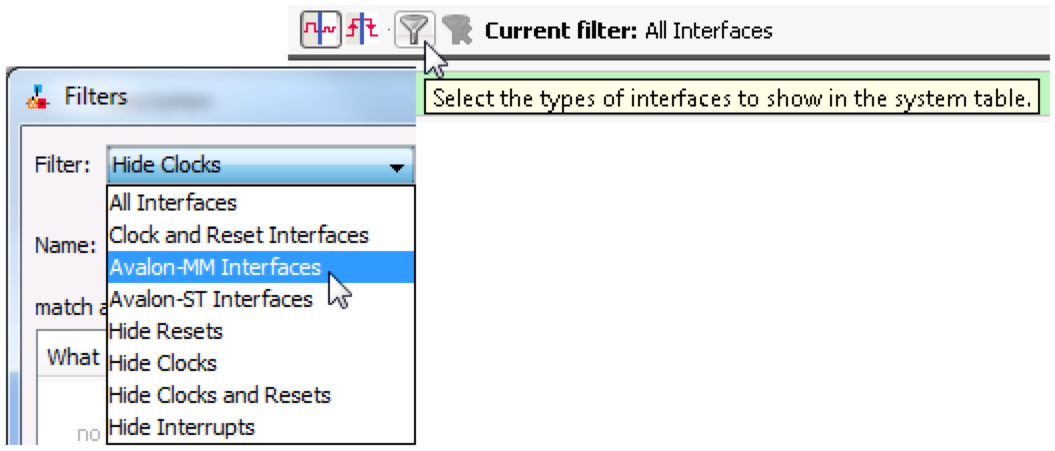
mm_bridge_0.m0
Connection from mm_bridge_0.m0 to mm_bridge_1.s0
● **A clock crossing adapter will be inserted**
mm_bridge_0.m0 *(source)*
Associated clock: clk_0.clk
mm_bridge_1.s0 *(sink)*
Associated clock: clk_1.clk

Filter Qsys Clock and Reset Domains in the System Contents Tab

You can filter the display of your Qsys clock and reset domains in the **System Contents** tab.

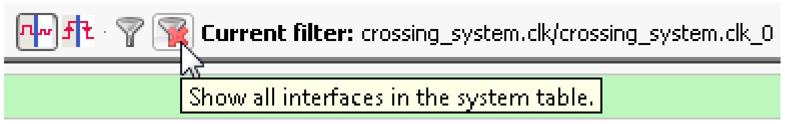
1. To filter the display in the **System Contents** tab to view only a particular interface and its connections, or to choose to hide clock, reset, or interrupt interfaces, click the **Filters** icon in the clock and reset control tool to open the **Filters** dialog box.
   The selected interfaces appear in the **System Contents** tab.

Figure 5-27: Filters Dialog Box



2. To clear all clock and reset filters in the **System Contents** tab and show all interfaces, click the **Filters** icon with the red "x" in the clock and reset control tool.

Figure 5-28: Show All Interfaces


Current filter: crossing_system.clk/crossing_system.clk_0
Show all interfaces in the system table.

Specify Qsys Interconnect Requirements

The **Interconnect Requirements** tab allows you to apply system-wide, `$system`, and interface interconnect requirements for IP components in your system. Options in the **Setting** column vary depending on what you select in the **Identifier** column

Table 5-3: Specifying System-Wide Interconnect Requirements

Option	Description
Limit interconnect pipeline stages to	Specifies the maximum number of pipeline stages that Qsys may insert in each command and response path to increase the f_{MAX} at the expense of additional latency. You can specify between 0–4 pipeline stages, where 0 means that the interconnect has a combinational data path. Choosing 3 or 4 pipeline stages may significantly increase the logic utilization of the system. This setting is specific for each Qsys system or subsystem, meaning that each subsystem can have a different setting. Additional latency is added once on the command path, and once on the response path. You can manually adjust this setting in the **Memory-Mapped Interconnect** tab. Access this tab by clicking **Show System With Qsys Interconnect command** on the System menu.

Option	Description
Clock crossing adapter type	Specifies the default implementation for automatically inserted clock crossing adapters: • **Handshake**—This adapter uses a simple hand-shaking protocol to propagate transfer control signals and responses across the clock boundary. This methodology uses fewer hardware resources because each transfer is safely propagated to the target domain before the next transfer can begin. The **Handshake** adapter is appropriate for systems with low throughput requirements. • **FIFO**—This adapter uses dual-clock FIFOs for synchronization. The latency of the FIFO-based adapter is a couple of clock cycles more than the handshaking clock crossing component. However, the FIFO-based adapter can sustain higher throughput because it supports multiple transactions at any given time. FIFO-based clock crossing adapters require more resources. The **FIFO** adapter is appropriate for memory-mapped transfers requiring high throughput across clock domains. • **Auto**—If you select **Auto**, Qsys specifies the **FIFO** adapter for bursting links, and the **Handshake** adapter for all other links.
Automate default slave insertion	Specifies whether you want Qsys to automatically insert a default slave for undefined memory region accesses during system generation.
Enable instrumentation	Allows you to choose the converter type that Qsys applies to each burst. • • **Generic converter (slower, lower area)**—Default. Controls all burst conversions with a single converter that is able to adapt incoming burst types. This results in an adapter that has lower f_{max}, but smaller area. • **Per-burst-type converter (faster, higher area)**—Controls incoming bursts with a particular converter, depending on the burst type. This results in an adapter that has higher f_{max}, but higher area. This setting is useful when you have AXI masters or slaves and you want a higher f_{max}.
Burst Adapter Implementation	Allows you to choose the converter type that Qsys applies to each burst. • **Generic converter (slower, lower area)**—Default. Controls all burst conversions with a single converter that is able to adapt incoming burst types. This results in an adapter that has lower f_{max}, but smaller area. • **Per-burst-type converter (faster, higher area)**—Controls incoming bursts with a particular converter, depending on the burst type. This results in an adapter that has higher f_{max}, but higher area. This setting is useful when you have AXI masters or slaves and you want a higher f_{max}.

Option	Description
Enable ECC protection	Specifies the default implementation for ECC protection for memory elements. Currently supports only Read Data FIFO (`rdata_FIFO`) instances.. • **FALSE**—Default. ECC protection is disabled for memory elements in the Qsys interconnect. • **TRUE**—ECC protection is enabled for memory elements. Qsys interconnect sends ECC errors that cannot be corrected as `DECODEERROR` (`DECERR`) on the Avalon response bus. This setting may increase logic utilization and cause lower f_{Max}, but provides additional protection against data corruption. **Note:** For more information about Error Correction Coding (ECC), refer to *Error Correction Coding in Qsys Interconnect*.

Table 5-4: Specifying Interface Interconnect Requirements

You can apply the following interconnect requirements when you select a component interface as the **Identifier** in the **Interconnect Requirements** tab, in the **All Requirements** table.

Option	Value	Description
Security	• Non-secure • Secure • Secure ranges • TrustZone-aware	After you establish connections between the masters and slaves, allows you to set the security options, as needed, for each master and slave in your system. **Note:** You can also set these values in the **Security** column in the **System Contents** tab.
Secure address ranges	Accepts valid address range.	Allows you to type in any valid address range.

For more information about HPS, refer to the *Cyclone V Device Handbook* in volume 3 of the *Hard Processor System Technical Reference Manual*.

Related Information

Error Correction Coding in Qsys Interconnect

Manage Qsys System Security

TrustZone is the security extension of the ARM®-based architecture. It includes secure and non-secure transactions designations, and a protocol for processing between the designations. TrustZone security support is a part of the Qsys interconnect.

The AXI `AxPROT` protection signal specifies a secure or non-secure transaction. When an AXI master sends a command, the `AxPROT` signal specifies whether the command is secure or non-secure. When an AXI slave receives a command, the `AxPROT` signal determines whether the command is secure or non-secure. Determining the security of a transaction while sending or receiving a transaction is a run-time protocol.

The Avalon specification does not include a protection signal as part of its specification. When an Avalon master sends a command, it has no embedded security and Qsys recognizes the command as non-secure. When an Avalon slave receives a command, it also has no embedded security, and the slave always accepts the command and responds.

AXI masters and slaves can be TrustZone-aware. All other master and slave interfaces, such as Avalon-MM interfaces, are non-TrustZone-aware. You can set compile-time security support for all components (except AXI masters, including AXI3, AXI4,and AXI4-Lite) in the **Security** column in the **System Contents** tab, or in the **Interconnect Requirements** tab under the **Identifier** column for the master or slave interface. To begin creating a secure system, you must first add masters and slaves to your system, and the connections between them. After you establish connections between the masters and slaves, you can then set the security options, as needed

An example of when you may need to specify compile-time security support is when an Avalon master needs to communicate with a secure AXI slave, and you can specify whether the connection point is secure or non-secure. You can specify a compile-time secure address ranges for a memory slave if an interface-level security setting is not sufficient.

Related Information

- Qsys Interconnect on page 7-1
- Qsys System Design Components on page 10-1

Configure Qsys Security Settings Between Interfaces

The AXI `AxPROT` signal specifies a transaction as secure or non-secure at runtime when a master sends a transaction. Qsys identifies AXI master interfaces as TrustZone-aware. You can configure AXI slaves as Trustzone-aware, secure, non-secure, or secure ranges.

Table 5-5: Compile-Time Security Options

For non-TrustZone-aware components, compile-time security support options are available in Qsys on the **System Contents** tab, or on the **Interconnect Requirements** tab.

Compile-Time Security Options	Description
Non-secure	Master sends only non-secure transactions, and the slave receives any transaction, secure or non-secure.
Secure	Master sends only secure transactions, and the slave receives only secure transactions.
Secure ranges	Applies to only the slave interface. The specified address ranges within the slave's address span are secure, all other address ranges are not. The format is a comma-separated list of inclusive-low and inclusive-high addresses, for example, `0x0:0xfff, 0x2000:0x20ff`.

After setting compile-time security options for non-TrustZone-aware master and slave interfaces, you must identify those masters that require a default slave before generation. To designate a slave interface as the default slave, turn on **Default Slave** in the **System Contents** tab. A master can have only one default slave.

Note: The **Security** and **Default Slave** columns in the **System Contents** tab are hidden by default. Right-click the **System Contents** header to select which columns you want to display.

The following are descriptions of security support for master and slave interfaces. These description can guide you in your design decisions when you want to create secure systems that have mixed secure and non-TrustZone-aware components:

- All AXI, AXI4, and AXI4-Lite masters are TrustZone-aware.
- You can set AXI, AXI4, and AXI4-Lite slaves as Trust-Zone-aware, secure, non-secure, or secure range ranges.
- You can set non-AXI master interfaces as secure or non-secure.
- You can set non-AXI slave interfaces as secure, non-secure, or secure address ranges.

Specify a Default Slave in a Qsys System

If a master issues "per-access" or "not allowed" transactions, your design must contain a default slave. Per-access refers to the ability of a TrustZone-aware master to allow or disallow access or transactions. A transaction that violates security is rerouted to the default slave and subsequently responds to the master with an error. You can designate any slave as the default slave.

You can share a default slave between multiple masters. You should have one default slave for each interconnect domain. An interconnect domain is a group of connected memory-mapped masters and slaves that share the same interconnect. The `altera_axi_default_slave` component includes the required TrustZone features.

You can achieve an optimized secure system by partitioning your design and carefully designating secure or non-secure address maps to maintain reliable data. Avoid a design where, under the same hierarchy, a non-secure master initiates transactions to a secure slave resulting in unsuccessful transfers.

Table 5-6: Secure and Non-Secure Access Between Master, Slave, and Memory Components

Transaction Type	TrustZone-aware Master	Non-TrustZone-aware Master Secure	Non-TrustZone-aware Master Non-Secure
TrustZone-aware slave/memory	OK	OK	OK
Non-TrustZone-aware slave (secure)	Per-access	OK	Not allowed
Non-TrustZone-aware slave (non-secure)	OK	OK	OK
Non-TrustZone-aware memory (secure region)	Per-access	OK	Not allowed

Transaction Type	TrustZone-aware Master	Non-TrustZone-aware Master Secure	Non-TrustZone-aware Master Non-Secure
Non-TrustZone-aware memory (non-secure region)	OK	OK	OK

Access Undefined Memory Regions

When a transaction from a master targets a memory region that is not specified in the slave memory map, it is known as an "access to an undefined memory region." To ensure predictable response behavior when this occurs, you must add a default slave to your design. Qsys then routes undefined memory region accesses to the default slave, which terminates the transaction with an error response.

You can designate any memory-mapped slave as a default slave. Altera recommends that you have only one default slave for each interconnect domain in your system. Accessing undefined memory regions can occur in the following cases:

- When there are gaps within the accessible memory map region that are within the addressable range of slaves, but are not mapped.
- Accesses by a master to a region that does not belong to any slaves that is mapped to the master.
- When a non-secured transaction is accessing a secured slave. This applies to only slaves that are secured at compilation time.
- When a read-only slave is accessed with a write command, or a write-only slave is accessed with a read command.

To designate a slave as the default slave, for the selected component, turn on **Default Slave** in the **Systems Content** tab.

Note: If you do not specify the default slave, Qsys automatically assigns the slave at the lowest address within the memory map for the master that issues the request as the default slave.

Related Information

- **Qsys System Design Components** on page 10-1

Integrate a Qsys System with the Quartus II Software

To integrate a Qsys system with your Quartus II project, you must add either the Qsys System File (**.qsys**) or the Quartus II IP File (**.qip**), but never both to your Quartus II project. Qsys creates the **.qsys** file when you save your Qsys system, and produces the **.qip** file when you generate your Qsys system. Both the **.qsys** and **.qip** files contain the information necessary for compiling your Qsys system within a Quartus II project.

You can choose to include the **.qsys** file automatically in your Quartus II project when you generate your Qsys system by turning on the **Automatically add Quartus II IP files to all projects** option in the Quartus II software (**Tools** > **Options** > **IP Settings**). If this option is turned off, the Quartus II software asks you if you want to include the **.qsys** file in your Quartus II project after you exit Qsys.

If you want file generation to occur as part of the Quartus II software's compilation, you should include the **.qsys** file in your Quartus II project. If you want to manually control file generation outside of the Quartus II software, you should include the **.qip** file in your Quartus II project.

Note: The Quartus II software generates an error message during compilation if you add both the **.qsys** and **.qip** files to your Quartus II project.

Does Quartus II Overwrite Qsys-Generated Files During Compilation?

Qsys supports standard and legacy device generation. Standard device generation refers to generating files for the Arria 10 device, and later device families. Legacy device generation refers to generating files for device families prior to the release of the Arria 10 device, including Max 10 devices.

When you integrate your Qsys system with the Quartus II software, if a **.qsys** file is included as a source file, Qsys generates standard device files under **<*system*>/** next to the location of the **.qsys** file. For legacy devices, if a **.qsys** file is included as a source file, Qsys generates HDL files in the Quartus II project directory under **/db/ip**.

For standard devices, Qsys-generated files are only overwritten during Quartus II compilation if the **.qip** file is removed or missing. For legacy devices, each time you compile your Quartus II project with a **.qsys** file, the Qsys-generated files are overwritten. Therefore, you should not edit Qsys-generated HDL in the **/db/ip** directory; any edits made to these files are lost and never used as input to the Quartus HDL synthesis engine.

Related Information

- **Add IP Components to the Qsys IP Catalog** on page 5-25
- **Generate a Qsys System** on page 5-54
- **Qsys Synthesis Standard and Legacy Device Output Directories** on page 5-58
- **Qsys Simulation Standard and Legacy Device Output Directories** on page 5-59
- **Introduction to Altera IP Cores**
- **Implementing and Parameterizing Memory IP**

Integrate a Qsys System and the Quartus II Software With the .qsys File

Use the following steps to integrate your Qsys system and your Quartus II project using the **.qsys** file:

1. In Qsys, create and save a Qsys system.
2. To automatically include the **.qsys** file in the your Quartus II project during compilation, in the Quartus II software, select **Tools** > **Options** > **IP Settings**, and turn on **Automatically add Quartus II IP files to all projects**.
3. When the **Automatically add Quartus II IP files to all projects** option is not checked, when you exit Qsys, the Quartus II software displays a dialog box asking whether you want to add the **.qsys** file to your Quartus II project. Click **Yes** to add the **.qsys** file to your Quartus II project.
4. In the Quartus II software, select **Processing** > **Start Compilation**.

Integrate a Qsys System and the Quartus II Software With the .qip File

Use the following steps to integrate your Qsys system and your Quartus II project using the **.qip** file:

1. In Qsys, create and save a Qsys system.
2. In Qsys, click **Generate HDL**.
3. In the Quartus II software, select **Assignments** > **Settings** > **Files**.
4. On the **Files** page, use the controls to locate your **.qip** file, and then add it to your Quartus II project.
5. In the Quartus II software, select **Processing** > **Start Compilation**.

Manage IP Settings in the Quartus II Software

To specify the following IP Settings in the Quartus II software, click **Tools** > **Option** > **IP Settings**:

Table 5-7: IP Settings

Setting	Description
Maximum Qsys memory usage	Allows you to increase memory usage for Qsys if you experience slow processing for large systems, or if Qsys reports an **Out of Memory** error.
IP generation HDL preference	The Quartus II software uses this setting when the **.qsys** file appears in the **Files** list for the current project in the **Settings** dialog box and you run Analysis & Synthesis. Qsys uses this setting when you generate HDL files.
Automatically add Quartus II IP files to all projects	The Quartus II software uses this setting when you create an IP core file variation with options in the Quartus II IP Catalog and parameter editor. When turned on, the Quartus II software adds the IP variation files to the project currently open.
IP Catalog Search Locations	The Quartus II software uses the settings that you specify for global and project search paths under **IP Search Locations**, in addition to the **IP Search Path** in Qsys (**Tools** > **Options**), to populate the Quartus II software IP Catalog. Qsys uses the uses the settings that you specify for global search paths under **IP Search Locations** to populate the Qsys IP Catalog, which appears in Qsys (**Tools** > **Options**). Qsys uses the project search path settings to populate the Qsys IP Catalog when you open Qsys from within the Quartus II software (**Tools** > **Qsys**), but not when you open Qsys from the command-line.

Note: You can also access **IP Settings** by clicking **Assignments** > **Settings** > **IP Settings**. This access is available only when you have a Quartus II project open. This allows you access to **IP Settings** when you want to create IP cores independent of a Quartus II project. Settings that you apply or create in either location are shared.

Opening Qsys with Additional Memory

If your Qsys system requires more than the 512 megabytes of default memory, you can increase the amount of memory either in the Quartus II software **Options** dialog box, or at the command-line.

- When you open Qsys from within the Quartus II software, you can increase memory for your Qsys system, by clicking **Tools** > **Options** > **IP Settings**, and then selecting the appropriate amount of memory with the **Maximum Qsys memory usage** option.
- When you open Qsys from the command-line, you can add an option to increase the memory. For example, the following `qsys-edit` command allows you to open Qsys with 1 gigabytes of memory.

```
qsys-edit --jvm-max-heap-size=1g
```

Set Qsys Clock Constraints

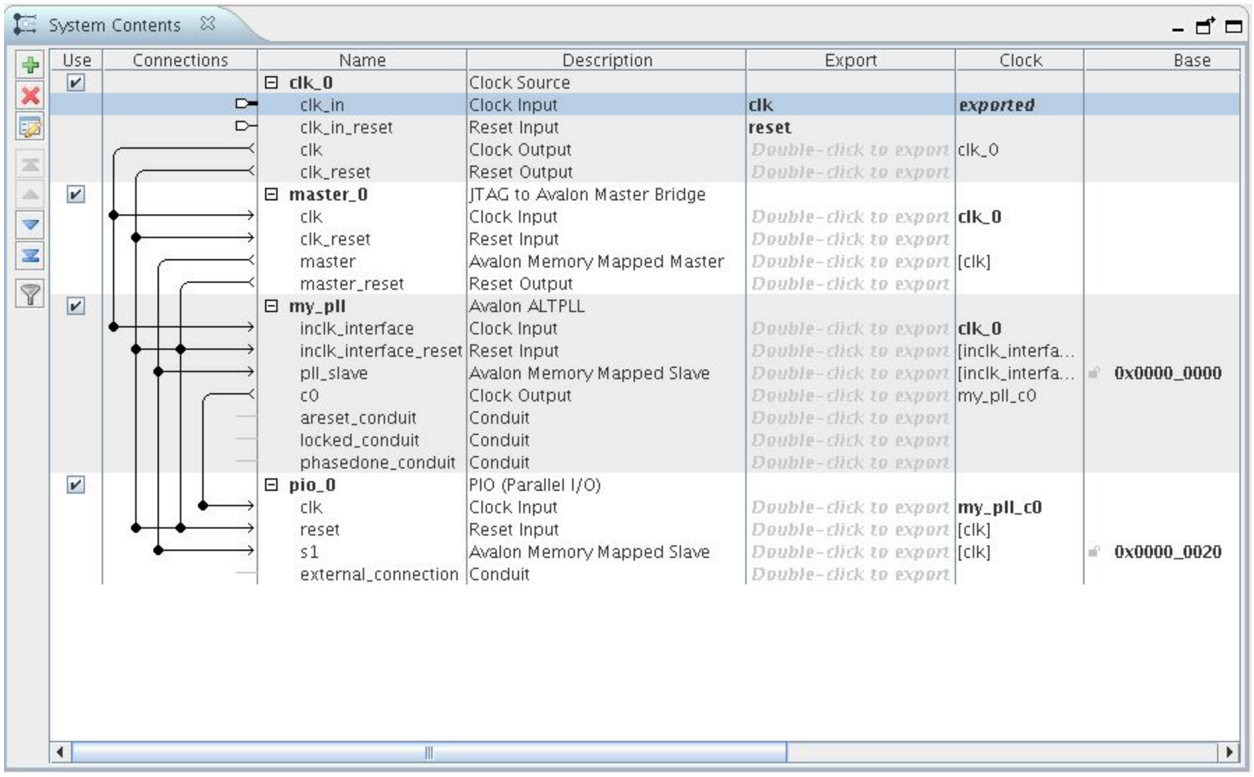
Many IP components include Synopsys Design Constraint (**.sdc**) files that provide timing constraints. Generated **.sdc** files are included in your Quartus II project with the generated **.qip** file. For your top-level clocks and PLLs, you must provide clock and timing constraints in SDC format to direct synthesis and fitting to optimize the design appropriately, and to evaluate performance against timing constraints.

You can specify a base clock assignment for each clock input in the TimeQuest GUI or with the `create_clock` command, and then use the `derive_pll_clocks` command to define the PLL clock output frequencies and phase shifts for all PLLs in the Quartus II project using the **.sdc** file.

Figure 5-29: Single Clock Input Signal

For the case of a single clock input signal called `clk`, and one PLL with a single output, you can use the following commands in your Synopsys Design Constraint (**.sdc**) file:

```
create_clock -name master_clk -period 20 [get_ports {clk}]
derive_pll_clocks
```



System Contents

Use	Connections	Name	Description	Export	Clock	Base
✓		⊟ **clk_0**	Clock Source			
		clk_in	Clock Input	clk	*exported*	
		clk_in_reset	Reset Input	reset		
		clk	Clock Output	Double-click to export	clk_0	
		clk_reset	Reset Output	Double-click to export		
✓		⊟ **master_0**	JTAG to Avalon Master Bridge			
		clk	Clock Input	Double-click to export	**clk_0**	
		clk_reset	Reset Input	Double-click to export		
		master	Avalon Memory Mapped Master	Double-click to export	[clk]	
		master_reset	Reset Output	Double-click to export		
✓		⊟ **my_pll**	Avalon ALTPLL			
		inclk_interface	Clock Input	Double-click to export	**clk_0**	
		inclk_interface_reset	Reset Input	Double-click to export	[inclk_interfa...	
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	[inclk_interfa...	**0x0000_0000**
		c0	Clock Output	Double-click to export	my_pll_c0	
		areset_conduit	Conduit	Double-click to export		
		locked_conduit	Conduit	Double-click to export		
		phasedone_conduit	Conduit	Double-click to export		
✓		⊟ **pio_0**	PIO (Parallel I/O)			
		clk	Clock Input	Double-click to export	**my_pll_c0**	
		reset	Reset Input	Double-click to export	[clk]	
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	**0x0000_0020**
		external_connection	Conduit	Double-click to export		

Related Information

The Quartus II TimeQuest Timing Analyzer

Generate a Qsys System

In Qsys, you can choose options for generation of synthesis, simulation and testbench files for your Qsys system.

Qsys system generation creates the interconnect between IP components and generates synthesis and simulation HDL files. You can generate a testbench system that adds Bus Functional Models (BFMs) that interact with your system in a simulator.

When you make changes to a system, Qsys gives you the option to exit without generating. If you choose to generate your system before you exit, the **Generation** dialog box opens and allows you to select generation options.

The **Generate HDL** button in the lower-right of the Qsys window allows you to quickly generate synthesis and simulation files for your system.

Note: If you cannot find the memory interface generated by Qsys when you use EMIF (External Memory Interface Debug Toolkit), verify that the **.sopcinfo** file appears in your Qsys project folder.

Related Information

- Avalon Verification IP Suite User Guide
- Mentor Verification IP (VIP) Altera Edition (AE)
- External Memory Interface Debug Toolkit

Set the Generation ID

The **Generation Id** parameter is a unique integer value that is set to a timestamp during Qsys system generation. System tools, such as NIOS II or HPS (Hard Processor System) use the **Generation ID** to ensure software-build compatibility with your Qsys system.

To set the **Generation Id** parameter, select the top-level system in the **Hierarchy** tab, and then locating the parameter in the open **Parameters** tab.

Generate Files for Synthesis and Simulation

The Quartus II software uses Qsys-generated synthesis HDL files during compilation.

In Qsys, you can generate simulation HDL files (**Generate** > **Generate HDL**), which can include simulation-only features targeted towards your simulator. You can generate simulation files as Verilog, VHDL, or as a mixed-language simulation for use in your simulation environment.

Note: For a list of Altera-supported simulators, refer to *Simulating Altera Designs*.

Qsys supports standard and legacy device generation. Standard device generation refers to generating files for the Arria 10 device, and later device families. Legacy device generation refers to generating files for device families prior to the release of the Arria 10 device, including Max 10 devices.

The **Output Directory** option applies to both synthesis and simulation generation. By default, the path of the generation output directory is fixed relative to the **.qsys** file. You can change the default directory in the **Generation** dialog box for legacy devices. For standard devices, the generation directory is fixed to the Qsys project directory.

Note: If you need to change top-level I/O pin or instance names, create a top-level HDL file that instantiates the Qsys system. The Qsys-generated output is then instantiated in your design without changes to the Qsys-generated output files.

The following options in the **Generation** dialog box (**Generate** > **Generate HDL**) allow you to generate synthesis and simulation files:

Option	Description
Create HDL design files for synthesis	Generates Verilog HDL or VHDL design files for the system's top-level definition and child instances for the selected target language. Synthesis file generation is optional.

Option	Description
Create timing and resource estimates for third-party EDA synthesis tools	Generates a non-functional Verilog Design File (**.v**) for use by some third-party EDA synthesis tools. Estimates timing and resource usage for your IP component. The generated netlist file name is **<your_ip_component_name>_syn.v**.
Create Block Symbol File (.bsf)	Allows you to optionally create a (**.bsf**) file to use in a schematic Block Diagram File (**.bdf**).
Create simulation model	Allows you to optionally generate Verilog HDL or VHDL simulation model files, and simulation scripts.
Allow mixed-language simulation	Generates a simulation model that contains both Verilog and VHDL as specified by the individual IP cores. Using this option, each IP core produces their HDL using it's native implementation, which results in simulation HDL that is easier to understand and faster to simulate. You must have a simulator that supports mixed language simulation.

Note: Modelsim-Altera now supports native mixed-language (VHDL/Verilog/SystemVerilog) simulation. Therefore, Altera simulation libraries may not be compatible with single language simulators. If you have a VHDL-only license, some versions of Mentor simulators may not be able to simulate IP written in Verilog. As a workaround, you can use Modelsim-Altera, or purchase a mixed language simulation license from Mentor.

Related Information

Simulating Altera Designs

Files Generated for Qsys IP Components

Qsys generates the following files for your Qsys system that are used during synthesis and simulation.

Table 5-8: Qsys-Generated IP Component Files

File Name	Description
<*system*>.qsys	The Qsys system file. <*system*> is the name that you give your system.
<*system*>.sopcinfo	Describes the connections and IP component parameterizations in your Qsys system. You can parse its contents to get requirements when you develop software drivers for IP components. Downstream tools such as the Nios II tool chain use this file. The **.sopcinfo** file and the **system.h** file generated for the Nios II tool chain include address map information for each slave relative to each master that accesses the slave. Different masters may have a different address map to access a particular slave component.

File Name	Description
<*system*>.cmp	The VHDL Component Declaration (**.cmp**) file is a text file that contains local generic and port definitions that you can use in VHDL design files.
<*system*>.html	A system report that contains connection information, a memory map showing the address of each slave with respect to each master to which it is connected, and parameter assignments.
<*system*>_generation.rpt	Qsys generation log file. A summary of the messages that Qsys issues during system generation.
<*system*>.debuginfo	Contains post-generation information. Used to pass System Console and Bus Analyzer Toolkit information about the Qsys interconnect. The Bus Analysis Toolkit uses this file to identify debug components in the Qsys interconnect.
<*system*>.qip	Contains all the required information about the IP component or system to integrate and compile the component in the Quartus II software.
<*system*>.bsf	A Block Symbol File (**.bsf**) representation of the top-level Qsys system for use in Quartus II Block Diagram Files (**.bdf**).
<*system*>.spd	Input file for `ip-make-simscript` to generate simulation scripts for supported simulators. The **.spd** file contains a list of files generated for simulation, including memory initialization files.
<*system*>.ppf	The Pin Planner File (**.ppf**) stores the port and node assignments for IP components created for use with the Pin Planner.
<*system*>_bb.v	You can use the Verilog black-box (**_bb.v**) file as an empty module declaration for use as a black box.
<*system*>.sip	Contains information required for NativeLink simulation of IP components. You must add the **.sip** file to your Quartus II project.
<*system*>_inst.v or **_inst.vhd**	HDL example instantiation template. You can copy and paste the contents of this file into your HDL file to instantiate a Qsys system.
<*system*>.regmap	If IP in the system contain register information, Qsys generates a **.regmap** file. The **.regmap** file describes the register map information of master and slave interfaces. This file complements the **.sopcinfo** file by providing more detailed register information about the system. This enables register display views and user customizable statistics in the System Console.

File Name	Description
<*system*>.svd	Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Qsys system. During synthesis, the **.svd** files for slave interfaces visible to System Console masters are stored in the **.sof** file in the debug section. System Console reads this section, which Qsys can query for register map information. For system slaves, Qsys can access the registers by name.
<*system*>.v or **<*system*>.vhd**	HDL files that instantiate each submodule or child IP core in the system for synthesis or simulation.
mentor/	Contains a ModelSim script **msim_setup.tcl** to set up and run a simulation.
aldec/	Contains a Riviera-PRO script **rivierapro_setup.tcl** to setup and run a simulation.
/synopsys/vcs **/synopsys/vcsmx**	Contains a shell script **vcs_setup.sh** to set up and run a VCS simulation. Contains a shell script **vcsmx_setup.sh** and **synopsys_ sim.setup** file to set up and run a VCS MX simulation.
/cadence	Contains a shell script **ncsim_setup.sh** and other setup files to set up and run an NCSIM simulation.
/submodules	Contains HDL files for the submodule of the Qsys system.
<*child IP cores*>/	For each generated child IP core directory, Qsys generates **/synth** and **/ sim** subdirectories.

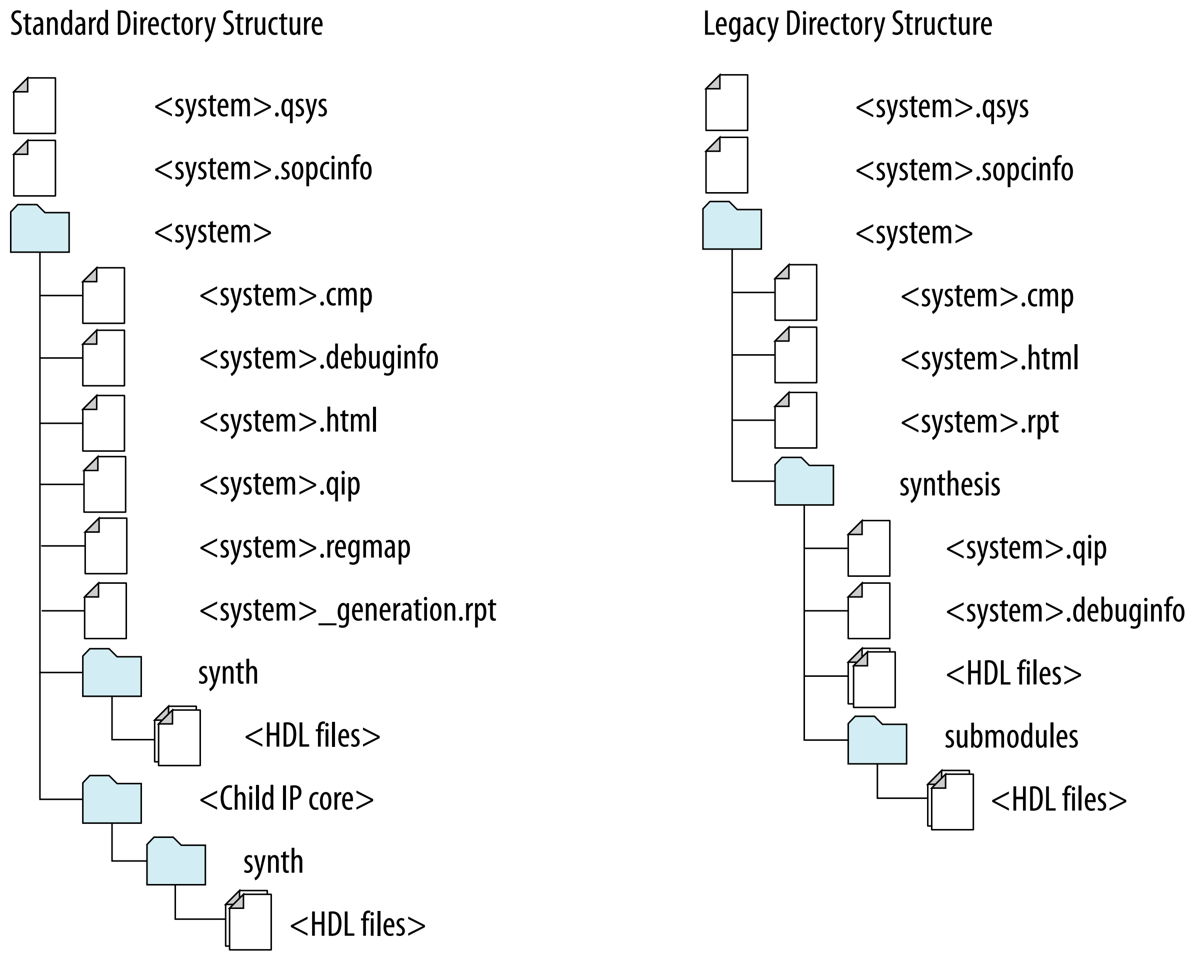
Related Information

- Qsys Synthesis Standard and Legacy Device Output Directories on page 5-58
- Qsys Simulation Standard and Legacy Device Output Directories on page 5-59

Qsys Synthesis Standard and Legacy Device Output Directories

The **/synth** or **/synthesis** directories contain the Qsys-generated files that the Quartus II software uses to synthesize your design.

Figure 5-30: Qsys Synthesis Output Directories



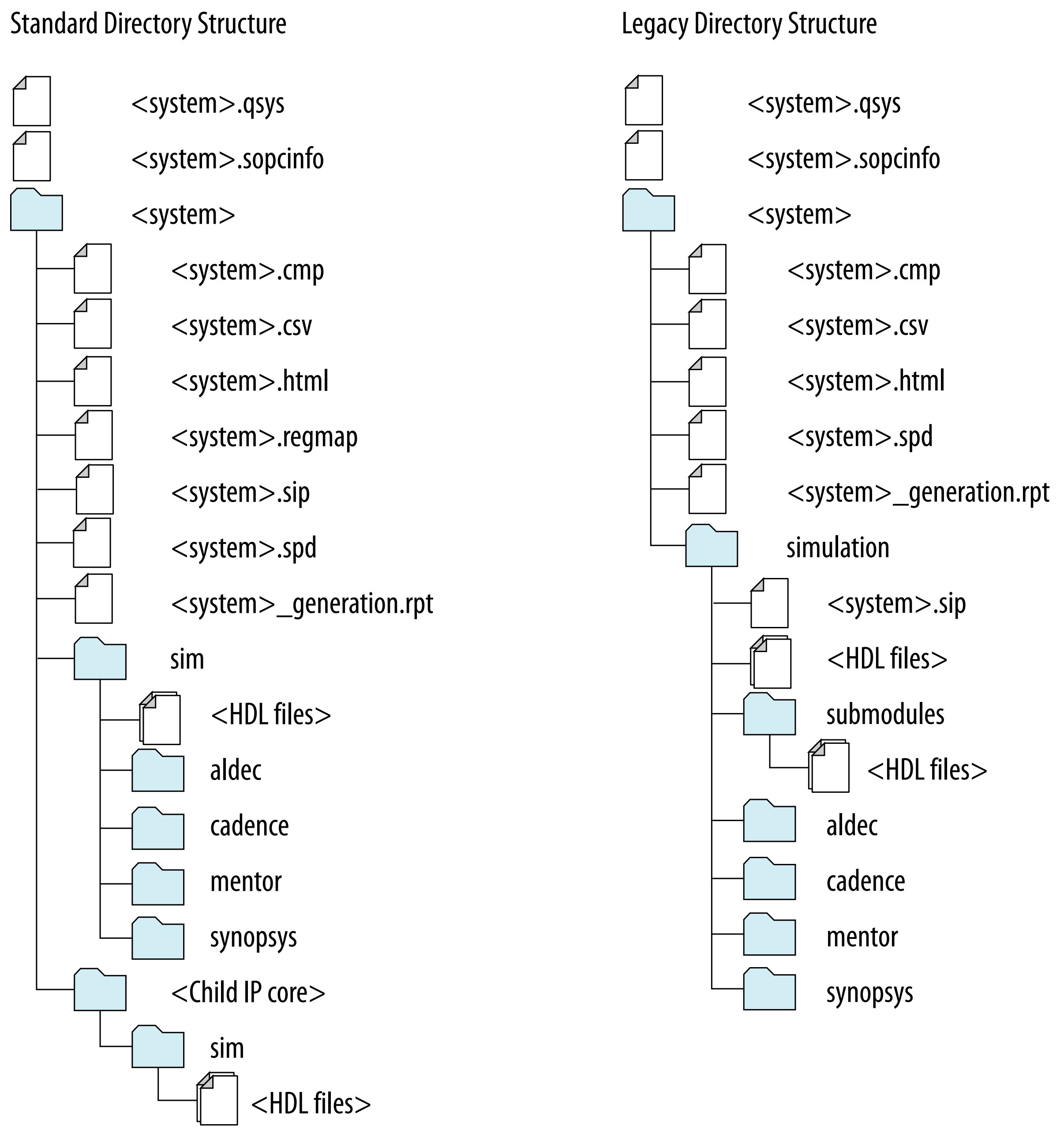
Related Information

Files Generated for Qsys IP Components on page 5-56

Qsys Simulation Standard and Legacy Device Output Directories

The **/sim** and **/simulation** directories contain the Qsys-generated output files to simulate your Qsys system.

Figure 5-31: Qsys Simulation Output Directories



Related Information

Files Generated for Qsys IP Components on page 5-56

Generate Files for a Testbench Qsys System

Qsys testbench is a new system that instantiates the current Qsys system by adding BFMs to drive the top-level interfaces. BFMs interact with the system in the simulator. You can use options in the **Generation** dialog box (**Generate** > **Generate Testbench System**) to generate a testbench Qsys system.

You can generate a standard or simple testbench system with BFM or Mentor Verification IP (for AXI3/AXI4) IP components that drive the external interfaces of your system. Qsys generates a Verilog HDL or VHDL simulation model for the testbench system to use in your simulation tool. You should first generate a testbench system, and then modify the testbench system in Qsys before generating its simulation model. In most cases, you should select only one of the simulation model options.

By default, the path of the generation output directory is fixed relative to the **.qsys** file. You can change the default directory in the **Generation** dialog box for legacy devices. For standard devices, the generation directory is fixed to the Qsys project directory.

The following options are available for generating a Qsys testbench system:

Option	Description
Create testbench Qsys system	• **Standard, BFMs for standard Qsys Interconnect**—Creates a testbench Qsys system with BFM IP components attached to exported Avalon and AXI3/AXI4 interfaces. Includes any simulation partner modules specified by IP components in the system. The testbench generator supports AXI interfaces and can connect AXI3/AXI4 interfaces to Mentor Graphics AXI3/AXI4 master/slave BFMs. However, BFMs support address widths only up to 32-bits. • **Simple, BFMs for clocks and resets**—Creates a testbench Qsys system with BFM IP components driving only clock and reset interfaces. Includes any simulation partner modules specified by IP components in the system.
Create testbench simulation model	Creates Verilog HDL or VHDL simulation model files and simulation scripts for the testbench Qsys system currently open in your workspace. Use this option if you do not need to modify the Qsys-generated testbench before running the simulation.
Allow mixed-language simulation	Generates a simulation model that contains both Verilog and VHDL as specified by the individual IP cores. Using this option, each IP core produces their HDL using it's native implementation, which results in simulation HDL that is easier to understand and faster to simulate. You must have a simulator that supports mixed language simulation.

Note: Modelsim-Altera now supports native mixed-language (VHDL/Verilog/SystemVerilog) simulation. Therefore, Altera simulation libraries may not be compatible with single language simulators. If you have a VHDL-only license, some versions of Mentor simulators may not be able to simulate IP written in Verilog. As a workaround, you can use Modelsim-Altera, or purchase a mixed language simulation license from Mentor.

Files Generated for Qsys Testbench

Table 5-9: Qsys-Generated Testbench Files

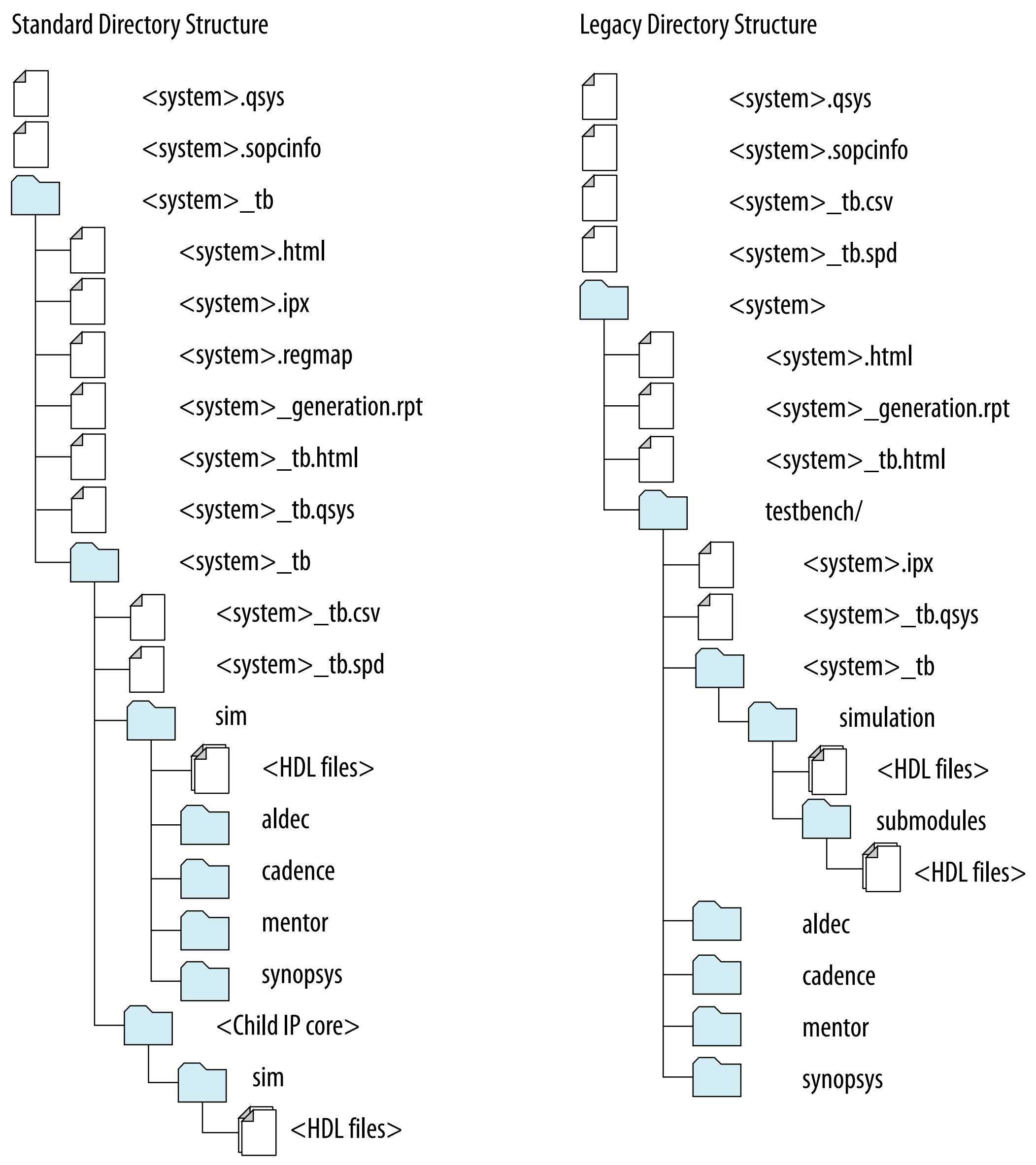
File Name or Directory Name	Description
<*system*>_tb.qsys	The Qsys testbench system.
<*system*>_tb.v or **<*system*>_tb.vhd**	The top-level testbench file that connects BFMs to the top-level interfaces of **<*system*>_*tb*.qsys**.
<*system*>_tb.spd	Required input file for `ip-make-simscript` to generate simulation scripts for supported simulators. The **.spd** file contains a list of files generated for simulation and information about memory that you can initialize.
<*system*>.html and **<*system*>_tb.html**	A system report that contains connection information, a memory map showing the address of each slave with respect to each master to which it is connected, and parameter assignments.
<*system*>_generation.rpt	Qsys generation log file. A summary of the messages that Qsys issues during testbench system generation.
<*system*>.ipx	The IP Index File (**.ipx**) lists the available IP components, or a reference to other directories to search for IP components.
<*system*>.svd	Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Qsys system. Similarly, during synthesis the **.svd** files for slave interfaces visible to System Console masters are stored in the **.sof** file in the debug section. System Console reads this section, which Qsys can query for register map information. For system slaves, Qsys can access the registers by name.
mentor/	Contains a ModelSim script **msim_setup.tcl** to set up and run a simulation
aldec/	Contains a Riviera-PRO script **rivierapro_setup.tcl** to setup and run a simulation.
/synopsys/vcs **/synopsys/vcsmx**	Contains a shell script **vcs_setup.sh** to set up and run a VCS simulation. Contains a shell script **vcsmx_setup.sh** and **synopsys_ sim.setup** file to set up and run a VCS MX simulation.
/cadence	Contains a shell script **ncsim_setup.sh** and other setup files to set up and run an NCSIM simulation.

File Name or Directory Name	Description
/submodules	Contains HDL files for the submodule of the Qsys testbench system.
<child IP cores>/	For each generated child IP core directory, Qsys testbench generates **/synth** and **/sim** subdirectories.

Qsys Testbench Simulation Standard and Legacy Device Output Directories

The **/sim** and **/simulation** directories contain the Qsys-generated output files to simulate your Qsys testbench system.

Figure 5-32: Qsys Simulation Testbench Directory Structure



Generate and Modify a Qsys Testbench System

You can use the following steps to create a Qsys testbench system of your Qsys system.

1. Create a Qsys system.
2. Generate a testbench system in the Qsys **Generation** dialog box (**Generate** > **Generate Testbench System**).
3. Open the testbench system in Qsys. Make changes to the BFMs, as needed, such as changing the instance names and **VHDL ID** value. For example, you can modify the **VHDL ID** value in the **Altera Avalon Interrupt Source** IP component.
4. If you modify a BFM, regenerate the simulation model for the testbench system.
5. Create a custom test program for the BFMs.
6. Compile and load the Qsys system and testbench into your simulator, and then run the simulation.

Simulation Scripts

Qsys generates simulation scripts to set up the simulation environment for Mentor Graphics Modelsim® and Questasim®, Synopsys VCS and VCS MX, Cadence Incisive Enterprise Simulator® (NCSIM), and the Aldec Riviera-PRO® Simulator.

You can use scripts to compile the required device libraries and system design files in the correct order and elaborate or load the top-level system for simulation.

Table 5-10: Simulation Script Variables

The simulation scripts provide variables that allow flexibility in your simulation environment.

Variable	Description
`TOP_LEVEL_NAME`	If the testbench Qsys system is not the top-level instance in your simulation environment because you instantiate the Qsys testbench within your own top-level simulation file, set the `TOP_LEVEL_NAME` variable to the top-level hierarchy name.
`QSYS_SIMDIR`	If the simulation files generated by Qsys are not in the simulation working directory, use the `QSYS_SIMDIR` variable to specify the directory location of the Qsys simulation files.
`QUARTUS_INSTALL_DIR`	Points to the Quartus installation directory that contains the device family library.

Example 5-4: Top-Level Simulation HDL File for a Testbench System

The example below shows the `pattern_generator_tb` generated for a Qsys system called `pattern_generator`. The top**.sv** file defines the top-level module that instantiates the `pattern_generator_tb` simulation model, as well as a custom SystemVerilog test program with BFM transactions, called `test_program`.

```
module top();
 pattern_generator_tb tb();
 test_program pgm();
endmodule
```

Note: The VHDL version of the Altera Tristate Conduit BFM is not supported in Synopsys VCS, NCSim, and Riviera-PRO in the Quartus II software version 14.0. These simulators do not support the VHDL protected type, which is used to implement the BFM. For a workaround, use a simulator that supports the VHDL protected type.

Note: Modelsim-Altera now supports native mixed-language (VHDL/Verilog/SystemVerilog) simulation. Therefore, Altera simulation libraries may not be compatible with single language simulators. If you have a VHDL-only license, some versions of Mentor simulators may not be able to simulate IP written in Verilog. As a workaround, you can use Modelsim-Altera, or purchase a mixed language simulation license from Mentor.

Related Information

- **ModelSim-Altera software, Mentor Graphics ModelSim support**
- **Synopsys VCS and VCS MX support**
- **Cadence Incisive Enterprise Simulator (IES) support**
- **Aldec Active-HDL and Rivera-PRO support**

Simulating Software Running on a Nios II Processor

To simulate the software in a system driven by a Nios II processor, generate the simulation model for the Qsys testbench system with the following steps:

1. In the **Generation** dialog box (**Generate** > **Generate Testbench System**), select **Simple, BFMs for clocks and resets**.
2. For the **Create testbench simulation model** option select **Verilog** or **VHDL**.
3. Click **Generate.**
4. Open the **Nios II Software Build Tools for Eclipse**.
5. Set up an application project and board support package (BSP) for the <*system*> **.sopcinfo** file.
6. To simulate, right-click the application project in Eclipse, and then click **Run as** > **Nios II ModelSim**. Sets up the ModelSim simulation environment, and compiles and loads the Nios II software simulation.
7. To run the simulation in ModelSim, type `run -all` in the ModelSim transcript window.
8. Set the ModelSim settings and select the Qsys Testbench Simulation Package Descriptor (**.spd**) file, < *system* > **_tb.spd**. The **.spd** file is generated with the testbench simulation model for Nios II designs and specifies the files required for Nios II simulation.

Related Information

- **Getting Started with the Graphical User Interface (Nios II)**
- **Getting Started from the Command-Line (Nios II)**

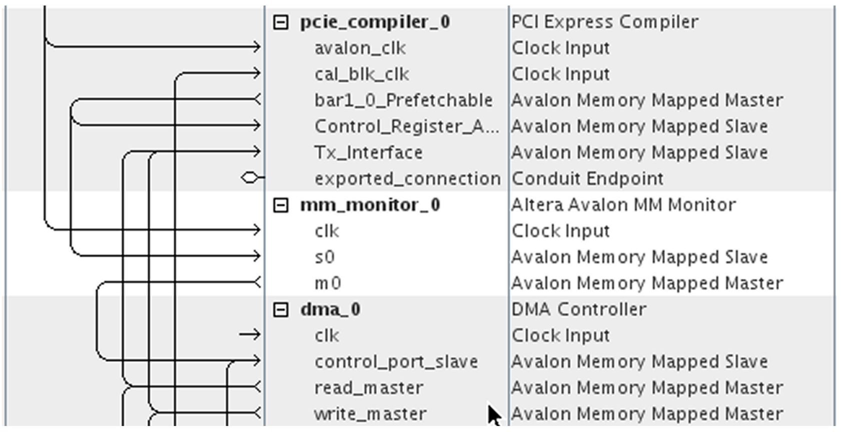
Add Assertion Monitors for Simulation

You can add monitors to Avalon-MM, AXI, and Avalon-ST interfaces in your system to verify protocol and test coverage with a simulator that supports SystemVerilog assertions.

Note: Modelsim Altera Edition does not support SystemVerilog assertions. If you want to use assertion monitors, you must use a supported third-party simulators such as Mentor Questasim, Synopsys VCS, or Cadence Incisive. For more information, refer to *Introduction to Altera IP Cores*.

Figure 5-33: Inserting an Avalon-MM Monitor Between an Avalon-MM Master and Slave Interface

This example demonstrates the use of a monitor with an Avalon-MM monitor between the `pcie_compiler bar1_0_Prefetchable` Avalon-MM master interface, and the `dma_0 control_port_slave` Avalon-MM slave interface.



⊟ pcie_compiler_0	PCI Express Compiler
avalon_clk	Clock Input
cal_blk_clk	Clock Input
bar1_0_Prefetchable	Avalon Memory Mapped Master
Control_Register_A...	Avalon Memory Mapped Slave
Tx_Interface	Avalon Memory Mapped Slave
exported_connection	Conduit Endpoint
⊟ mm_monitor_0	Altera Avalon MM Monitor
clk	Clock Input
s0	Avalon Memory Mapped Slave
m0	Avalon Memory Mapped Master
⊟ dma_0	DMA Controller
clk	Clock Input
control_port_slave	Avalon Memory Mapped Slave
read_master	Avalon Memory Mapped Master
write_master	Avalon Memory Mapped Master

Similarly, you can insert an Avalon-ST monitor between Avalon-ST source and sink interfaces.

Related Information

Introduction to Altera IP Cores

CMSIS Support for the HPS IP Component

Qsys systems that contain an HPS IP component generate a System View Description (**.svd**) file that lists peripherals connected to the ARM processor.

The **.svd** (or CMSIS-SVD) file format is an XML schema specified as part of the Cortex Microcontroller Software Interface Standard (CMSIS) provided by ARM. The **.svd** file allows HPS system debug tools (such as the DS-5 Debugger) to view the register maps of peripherals connected to HPS in a Qsys system.

Related Information

Component Interface Tcl Reference on page 9-1

CMSIS - Cortex Microcontroller Software

Explore and Manage Qsys Interconnect

The System with Qsys Interconnect window allows you to see the contents of the Qsys interconnect before you generate your system. In this display of your system, you can review a graphical representation of the generated interconnect. Qsys converts connections between interfaces to interconnect logic during system generation.

You access the System with Qsys Interconnect window by clicking **Show System With Qsys Interconnect** command on the System menu.

The System with Qsys Interconnect window has the following tabs:

- **System Contents**—Displays the original instances in your system, as well as the inserted interconnect instances. Connections between interfaces are replaced by connections to interconnect where applicable.
- **Hierarchy**—Displays a system hierarchical navigator, expanding the system contents to show modules, interfaces, signals, contents of subsystems, and connections.
- **Parameters**—Displays the parameters for the selected element in the **Hierarchy** tab.
- **Memory-Mapped Interconnect**—Allows you to select a memory-mapped interconnect module and view its internal command and response networks. You can also insert pipeline stages to achieve timing closure.

The **System Contents**, **Hierarchy**, and **Parameters** tabs are read-only. Edits that you apply on the **Memory-Mapped Interconnect** tab are automatically reflected on the **Interconnect Requirements** tab.

The **Memory-Mapped Interconnect** tab in the System with Qsys Interconnect window displays a graphical representation of command and response data paths in your system. Data paths allow you precise control over pipelining in the interconnect. Qsys displays separate figures for the command and response data paths. You can access the data paths by clicking their respective tabs in the **Memory-Mapped Interconnect** tab.

Each node element in a figure represents either a master or slave that communicates over the interconnect, or an interconnect sub-module. Each edge is an abstraction of connectivity between elements, and its direction represents the flow of the commands or responses.

Click **Highlight Mode** (**Path**, **Successors**, **Predecessors**) to identify edges and data paths between modules. Turn on **Show Pipeline Locations** to add greyed-out registers on edges where pipelining is allowed in the interconnect.

Note: You must select more than one module to highlight a path.

Manually Controlling Pipelining in the Qsys Interconnect

The **Memory-Mapped Interconnect** tab allows you to manipulate pipleline connections in the Qsys interconnect. You access the **Memory-Mapped Interconnect** tab by clicking the **Show System With Qsys Interconnect** command on the System menu.

Note: To increase interconnect frequency, you should first try increasing the value of the **Limit interconnect pipeline stages to** option on the **Interconnect Requirements** tab. You should only consider manually pipelining the interconnect if changes to this option do not improve frequency, and you have tried all other options to achieve timing closure, including the use of a bridge. Manually pipelining the interconnect should only be applied to complete systems.

1. In the **Interconnect Requirements** tab, first try increasing the value of the **Limit interconnect pipeline stages to** option until it no longer gives significant improvements in frequency, or until it causes unacceptable effects on other parts of the system.
2. In the Quartus II software, compile your design and run timing analysis.
3. Using the timing report, identify the critical path through the interconnect and determine the approximate mid-point. The following is an example of a timing report:

```
2.800 0.000 cpu_instruction_master|out_shifter[63]|q
3.004 0.204 mm_domain_0|addr_router_001|Equal5~0|datac
3.246 0.242 mm_domain_0|addr_router_001|Equal5~0|combout
3.346 0.100 mm_domain_0|addr_router_001|Equal5~1|dataa
3.685 0.339 mm_domain_0|addr_router_001|Equal5~1|combout
```

```
    4.153 0.468 mm_domain_0|addr_router_001|src_channel[5]~0|datad
    4.373 0.220 mm_domain_0|addr_router_001|src_channel[5]~0|combout
```

4. In Qsys, click **System** > **Show System With Qsys Interconnect**.
5. In the **Memory-Mapped Interconnect** tab, select the interconnect module that contains the critical path. You can determine the name of the module from the hierarchical node names in the timing report.
6. Click **Show Pipelinable Locations**. Qsys display all possible pipeline locations in the interconnect. Right-click the possible pipeline location to insert or remove a pipeline stage.
7. Locate the possible pipeline location that is closest to the mid-point of the critical path. The names of the blocks in the memory-mapped interconnect tab correspond to the module instance names in the timing report.
8. Right-click the location where you want to insert a pipeline, and then click **Insert Pipeline**.
9. Regenerate the Qsys system, recompile the design, and then rerun timing analysis. If necessary, repeat the manual pipelining process again until timing requirements are met.

Manual pipelining has the following limitations:

- If you make changes to your original system's connectivity after manually pipelining an interconnect, your inserted pipelines may become invalid. Qsys displays warning messages when you generate your system if invalid pipeline stages are detected. You can remove invalid pipeline stages with the **Remove Stale Pipelines** option in the **Memory-Mapped Interconnect** tab. Altera recommends that you do not make changes to the system's connectivity after manual pipeline insertion.
- Review manually-inserted pipelines when upgrading to newer versions of Qsys. Manually-inserted pipelines in one version of Qsys may not be valid in a future version.

Related Information

Specify Qsys $system Interconnect Requirements

Qsys System Design Components on page 10-1

Implement Performance Monitoring

You use the Qsys **Instrumentation** tab in to set up real-time performance monitoring using throughput metrics such as read and write transfers. The **Add debug instrumentation to the Qsys Interconnect** option allows you to interact with the Bus Analyzer Toolkit, which you can access on the Tools menu in the Quartus II software.

Qsys supports performance monitoring for only Avalon-MM interfaces. In your Qsys system, you can monitor the performance of no less than three, and no greater than 15 components at one time. The performance monitoring feature works with Quartus II software devices 13.1 and newer.

Note: For more information about the Bus Analyzer Toolkit and the Qsys `Instrumentation` tab, refer to the **Bus Analyzer Toolkit** page.

Related Information

Bus Analyzer Toolkit

Qsys 64-Bit Addressing Support

Qsys interconnect supports up to 64-bit addressing for all Qsys interfaces and IP components, with a range of: `0x0000 0000 0000 0000` to `0xFFFF FFFF FFFF FFFF`, inclusive.

Address parameters appear in the **Base** and **End** columns in the **System Contents** tab, on the **Address Map** tab, in the parameter editor, and in validation messages. Qsys displays as many digits as needed in order to display the top-most set bit, for example, 12 hex digits for a 48-bit address.

A Qsys system can have multiple 64-bit masters, with each master having its own address space. You can share slaves between masters and masters can map slaves to different addresses. For example, one master can interact with slave `0` at base address 0000_0000_0000, and another master can see the same slave at base address `c000_000_000`.

Quartus II debugging tools provide access to the state of an addressable system via the Avalon-MM interconnect. These are also 64-bit compatible, and process within a 64-bit address space, including a JTAG to Avalon master bridge.

For more information on design practices when using slaves with large address spans, refer to *Address Span Extender* in volume 1 of the *Quartus II Handbook*.

Related Information

- **Qsys System Design Components** on page 10-1

Support for Avalon-MM Non-Power of Two Data Widths

Qsys requires that you connect all multi-point Avalon-MM connections to interfaces with data widths that are equal to powers of two.

Qsys issues a validation error if an Avalon-MM master or slave interface on a multi-point connection is parameterized with a non-power of two data width.

Note: Avalon-MM point-to-point connections between an Avalon-MM master and an Avalon-MM slave are an exception and may set their data widths to a non-power of two.

View the Qsys HDL Example

Click **Generate** > **HDL Example** to generate a template for the top-level HDL definition of your Qsys system in either Verilog HDL or VHDL. The HDL template displays the IP component declaration.

You can copy and paste the example into a top-level HDL file that instantiates the Qsys system, if the Qsys system is not the top-level module in your Quartus II project.

Qsys System Example Designs

Click the **Example Design** button in the parameter editor to generate an example design.

If there are multiple example designs for an IP component, then there is a button for each example in the parameter editor. When you click the **Example Design** button, the **Select Example Design Directory** dialog box appears, where you can select the directory to save the example design.

The **Example Design** button does not appear in the parameter editor if there is no example. For some IP components, you can click **Generate** > **Example Designs** to access an example design.

The following Qsys system example designs demonstrate various design features and flows that you can replicate in your Qsys system.

Related Information

- NIOS II Qsys Example Design
- PCI Express Avalon-ST Qsys Example Design
- Triple Speed Ethernet Qsys Example Design

Qsys Command-Line Utilities

You can perform many of the functions available in the Qsys GUI from the command-line with the `qsys-edit`, `qsys-generate`, and `qsys-script` utilities.

You run Qsys command-line executables from the Quartus II installation directory:

<*Quartus II installation directory*>**\quartus\sopc_builder\bin**

You can use `qsys-generate` to generate a Qsys system or IP variation outside of the Qsys GUI. You can use `qsys-script` to create, manipulate or manage a Qsys system with command-line scripting.

For command-line help listing all options for these executables, type the following command:

<*Quartus II installation directory*>**\quartus\sopc_builder\bin**\<*executable name*> --help

Example 5-5: Qsys Command-Line Scripting Example

```
qsys-script --script=my_script.tcl \
--system-file=fancy.qsys my_script.tcl contains:
package require -exact qsys 13.1
# get all instance names in the system and print one by one
set instances [ get_instances ]
foreach instance $instances {
    send_message Info "$instance"
}
```

Note: You must add **$QUARTUS_ROOTDIR/sopc_builder/bin/** to the `PATH` variable to access command-line utilities. Once you add this `PATH` variable, you can launch the unities from any directory location.

Related Information

- Working with Instance Parameters in Qsys
- Altera Wiki Qsys Scripts

Run the Qsys Editor with qsys-edit

You can use the `qsys-edit` utility to run the Qsys editor from the command-line.

You can use the following options with the `qsys-edit` utility:

Table 5-11: qsys-edit Command-Line Options

Option	Usage	Description
`1st arg file`	Optional	The name of the **.qsys** system or **.qvar** variation file to edit.
`--search-path[=<`*`value`*`>]`	Optional	If omitted, Qsys uses a standard default path. If provided, Qsys searches a comma-separated list of paths. To include the standard path in your replacement, use "`$`", for example: `/extra/dir.$`.
`--project-directory=<`*`directory`*`>`	Optional	Allows you to find IP components in certain locations relative to the project, if any. By default, the current directory is:`'.'` . To exclude any project directory, use `''`.
`--new-component-type=<`*`value`*`>`	Optional	Allows you to specify the kind of instance that is parameterized in a variation.
`--debug`	Optional	Enables debugging features and output.
`--host-controller`	Optional	Launches the application with an XML host controller interface on standard input/output.
`--jvm-max-heap-size=<`*`value`*`>`	Optional	The maximum memory size Qsys uses for allocations when running `qsys-edit`. You specify this value as `<`*`size`*`><`*`unit`*`>`, where unit is `m` (or `M`) for multiples of megabytes, or `g` (or `G`) for multiples of gigabytes. The default value is 512`m`.
`--help`	Optional	Display help for `qsys-edit`.

Scripting IP Core Generation

You can alternatively use command-line utilities to define and generate an IP core variation outside of the Quartus II GUI.Use `qsys-script` to run a Tcl file that parameterizes an IP variation you define in a script. Then, use `qsys-generate` to generate a **.qsys** file representing your parameterized IP variation.

The `qsys-generate` command is the same as when generating using the Qsys GUI. For command-line help listing all options for these executables, type <*executable name*> `--help`

To create a instance of a parameterizable Altera IP core at the command line, rather than using the GUI, follow these steps:

1. Run `qsys-script` to execute a Tcl script, similar to the example, that instantiates the IP and sets the IP parameters defined by the script:

   ```
   qsys-script --script=<script_file>.tcl
   ```

2. Run `qsys-generate` to generate the RTL for the IP variation:

   ```
   qsys-generate <IP variation file>.qsys
   ```

Note: Creating an IP generation script is an advanced feature that requires access to special IP core parameters. For more information about creating an IP generation script, contact your Altera sales representative.

Table 5-12: qsys-generate Command-Line Options

Option	Usage	Description
`<1st arg file>`	Required	The name of the **.qsys** system file to generate.
`--synthesis=<VERILOG|VHDL>`	Optional	Creates synthesis HDL files that Qsys uses to compile the system in a Quartus II project. You must specify the preferred generation language for the top-level RTL file for the generated Qsys system.
`--block-symbol-file`	Optional	Creates a Block Symbol File (**.bsf**) for the Qsys system.
`--simulation=<VERILOG|VHDL>`	Optional	Creates a simulation model for the Qsys system. The simulation model contains generated HDL files for the simulator, and may include simulation-only features. You must specify the preferred simulation language.
`--testbench=<SIMPLE|STANDARD>`	Optional	Creates a testbench system that instantiates the original system, adding bus functional models (BFMs) to drive the top-level interfaces. When you generate the system, the BFMs interact with the system in the simulator.
`--testbench-simulation=<VERILOG|VHDL>`	Optional	After you create the testbench system, you can create a simulation model for the testbench system.

Option	Usage	Description
`--search-path=<value>`	Optional	If you omit this command, Qsys uses a standard default path. If you provide this command, Qsys searches a comma-separated list of paths. To include the standard path in your replacement, use "`$`", for example, "`/extra/dir,$`".
`--jvm-max-heap-size=<value>`	Optional	The maximum memory size that Qsys uses for allocations when running `qsys-generate`. You specify the value as *<size>* *<unit>*, where `unit` is m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512m.
`--family=<value>`	Optional	Specifies the device family.
`--part=<value>`	Optional	Specifies the device part number. If set, this option overrides the `--family` option.
`--allow-mixed-language-simulation`	Optional	Enables a mixed language simulation model generation. If true, if a preferred simulation language is set, Qsys uses a `fileset` of the component for the simulation model generation. When false, which is the default, Qsys uses the language specified with `--file-set=<value>` for all components for simulation model generation. The current version of the ModelSim-Altera simulator supports mixed language simulation.

Display Available IP Components with ip-catalog

The `ip-catalog` command displays a list of available IP components relative to the current Quartus II project directory. Use the following format for the `ip-catalog` command:

```
ip-catalog
            [--project-dir=<directory>]
   [--name=<value>]
   [--verbose]
   [--xml]
   [--help]
```

Table 5-13: ip-catalog Command-Line Options

Option	Usage	Description
`--project-dir=<directory>`	Optional	Finds IP components relative to the Quartus II project directory. By default, Qsys uses '.' as the current directory. To exclude a project directory, leave the value empty.
`--name=<value>`	Optional	Provides a pattern to filter the names of the IP components found. To show all IP components, use a * or ' '. By default, Qsys shows all IP components. The argument is not case sensitive.
`--verbose`	Optional	Reports the progress of the command.
`--xml`	Optional	Generates the output in XML format, in place of colon-delimited format.
`--help`	Optional	Displays help for the `ip-catalog` command.

Create an .ipx File with ip-make-ipx

The `ip-make-ipx` command creates an **.ipx** file and is a convenient way to include a collection of IP components from an arbitrary directory. You can edit the **.ipx** file to disable visibility of one or more IP components in the IP Catalog. Use the following format for the `ip-make-ipx` command:

```
ip-make-ipx
          [--source-directory=<directory>]
          [--output=<file>]
          [--relative-vars=<value>]
    [--thorough-descent]
          [--message-before=<value>]
          [--message-after=<value>]
                [--help]
```

Table 5-14: ip-make-ipx Command-Line Options

Option	Usage	Description
`--source-directory=<directory>`	Optional	Specifies the root directory for IP component files. The default directory is *.*. You can provide a comma-separated list of directories.
`--output=<file>`	Optional	Specifies the name of the file to generate. The default name is **/component.ipx**.
`--relative-vars=<value>`	Optional	Causes the output file to include references relative to the specified variable(s) where possible. You can specify multiple variables as a comma-separated list.

Option	Usage	Description
`--thorough-descent`	Optional	If set, a component or **.ipx** file in a directory does not stop Qsys from searching subdirectories.
`--message-before=<value>`	Optional	Prints a message: `stdout` when indexing begins.
`--message-after=<value>`	Optional	Sends a message: `stdout` when indexing is done.
`--help`	Optional	Displays help for the `ip-make-ipx` command.

Related Information

Set up the IP Index File (.ipx) to Search for IP Components on page 5-28

Generate a Qsys System with qsys-script

You can use the `qsys-script` utility to create and manipulate a Qsys system with Tcl scripting commands.

Note: You must provide a package version for the `qsys-script`. If you do not specify the `--package-version=<value>` command, you must then provide a Tcl script and request the system scripting API directly with the `package require -exact qsys` < `version` > command.

Table 5-15: qsys-script Command-Line Options

Option	Usage	Description
`--system-file=<file>`	Optional	Specifies the path to a **.qsys** file. Qsys loads the system before running scripting commands.
`--script=<file>`	Optional	A file that contains Tcl scripting commands that you can use to create or manipulate Qsys systems. If you specify both `--cmd` and `--script,` Qsys runs the `--cmd` commands before the script specified by `--script`.
`--cmd=<value>`	Optional	A string that contains Tcl scripting commands that you can use to create or manipulate a Qsys system. If you specify both `--cmd` and `--script,` Qsys runs the `--cmd` commands before the script specified by `--script`.
`--package-version=<value>`	Optional	Specifies which Tcl API scripting version to use and determines the functionality and behavior of the Tcl commands. The Quartus II software supports Tcl API scripting commands. If you do not specify the version on the command-line, your script must request the scripting API directly with the `package require -exact qsys` <*version*> command.

Option	Usage	Description
`--search-path=<value>`	Optional	If you omit this command, a Qsys uses a standard default path. If you provide this command, Qsys searches a comma-separated list of paths. To include the standard path in your replacement, use `"$"`, for example, `/< directory path >/dir,$`. Separate multiple directory references with a comma.
`--jvm-max-heap-size=<value>`	Optional	The maximum memory size that the `qsys-script` tool uses. You specify this value as `<size><unit>`, where unit is `m` (or `M`) for multiples of megabytes, or `g` (or `G`) for multiples of gigabytes.
`--help`	Optional	Displays help for the `qsys-script` utility.

Qsys Scripting Command Reference

The following are Qsys scripting commands:

set_instance_parameter_value on page 5-146

set_instance_property on page 5-147

set_interconnect_requirement on page 5-148

set_interface_property on page 5-149

set_module_property on page 5-150

set_project_property on page 5-151

set_use_testbench_naming_pattern on page 5-152

set_validation_property on page 5-153

unlock_avalon_base_address on page 5-154

validate_connection on page 5-155

validate_instance on page 5-156

validate_instance_interface on page 5-157

validate_system on page 5-158

add_connection

Description

Connects the named interfaces using an appropriate connection type. Both interface names consist of a child instance name, followed by the name of an interface provided by that module. For example, `mux0.out` is the interface named on the instance named `mux0`. Be careful to connect the start to the end, and not the reverse.

Usage

`add_connection` *<start>* [*<end>*]

Returns

No return value.

Arguments

start

The start interface that is connected, in <*`instance_name`*>.<*`interface_name`*> format. If the `end` argument is omitted, the connection must be of the form <*`instance1`*>.<*`interface`*>/<*`instance2`*>.<*`interface`*>.

end (optional)

The end interface that is connected, <*`instance_name`*>.<*`interface_name`*>.

Example

```
add_connection dma.read_master sdram.s1
```

Related Information

- **get_connection_parameter_value** on page 5-99
- **get_connection_property** on page 5-102
- **get_connections** on page 5-103
- **remove_connection** on page 5-139
- **set_connection_parameter_value** on page 5-145

add_instance

Description

Adds an instance of a component, referred to as a *child* or *child instance*, to the system.

Usage

`add_instance` <*name*> <*type*> [<*version*>]

Returns

No return value.

Arguments

name

Specifies a unique local name that you can use to manipulate the instance. Qsys uses this name in the generated HDL to identify the instance.

type

Refers to a kind of instance available in the IP Catalog, for example `altera_avalon_uart`.

version (optional)

The required version of the specified instance type. If no version is specified, Qsys uses the latest version.

Example

```
add_instance uart_0 altera_avalon_uart
```

Related Information

- **get_instance_parameter_value** on page 5-118
- **get_instance_property** on page 5-122
- **get_instances** on page 5-123
- **remove_instance** on page 5-141
- **set_instance_parameter_value** on page 5-146

add_interface

Description

Adds an interface to your system, which Qsys uses to export an interface from within the system. You specify the exported internal interface with `set_interface_property` *`<interface>`* `EXPORT_OF` *`instance.interface`*.

Usage

`add_interface` *<name> <type> <direction>*.

Returns

No return value.

Arguments

name

The name of the interface that Qsys exports from the system.

type

The type of interface.

direction

The interface direction.

Example

```
add_interface my_export conduit end
set_interface_property my_export EXPORT_OF uart_0.external_connection
```

Related Information

- **get_interface_ports** on page 5-127
- **get_interface_properties** on page 5-128
- **get_interface_property** on page 5-129
- **set_interface_property** on page 5-149

apply_preset

Description

Applies the settings in a preset to the specified instance.

Usage

`apply_preset` *<instance> <preset_name>*

Returns

No return value.

Arguments

instance

The name of the instance.

preset_name

The name of the preset.

Example

```
apply_preset cpu_0 "Custom Debug Settings"
```

auto_assign_base_addresses

Description

Assigns base addresses to all memory mapped interfaces on an instance in the system. Instance interfaces that are locked with `lock_avalon_base_address` keep their addresses during address auto-assignment.

Usage

`auto_assign_base_addresses` *<instance>*

Returns

No return value.

Arguments

instance

The name of the instance with memory mapped interfaces.

Example

```
auto_assign_base_addresses sdram
```

Related Information

- **auto_assign_system_base_addresses** on page 5-85
- **lock_avalon_base_address** on page 5-138
- **unlock_avalon_base_address** on page 5-154

auto_assign_system_base_addresses

Description

Assigns legal base addresses to all memory mapped interfaces on all instances in the system. Instance interfaces that are locked with `lock_avalon_base_address` keep their addresses during address auto-assignment.

Usage

```
auto_assign_system_base_addresses
```

Returns

No return value.

Arguments

No arguments.

Example

```
auto_assign_system_base_addresses
```

Related Information

- **auto_assign_base_addresses** on page 5-84
- **lock_avalon_base_address** on page 5-138
- **unlock_avalon_base_address** on page 5-154

auto_assign_irqs

Description

Assigns interrupt numbers to all connected interrupt senders on an instance in the system.

Usage

`auto_assign_irqs` *<instance>*

Returns

No return value.

Arguments

instance

The name of the instance with an interrupt sender.

Example

```
auto_assign_irqs uart_0
```

auto_connect

Description

Creates connections from an instance or instance interface to matching interfaces in other instances in the system. For example, Avalon-MM slaves connect to Avalon-MM masters.

Usage

`auto_connect` *<element>*

Returns

No return value.

Arguments

element

The name of the instance interface, or the name of an instance.

Example

```
auto_connect sdram
auto_connect uart_0.s1
```

Related Information

add_connection on page 5-80

create_system

Description

Replaces the current system with a new system with the specified name.

Usage

`create_system` [<*name*>]

Returns

No return value.

Arguments

name (optional)

The name of the new system.

Example

```
create_system my_new_system_name
```

Related Information

- **load_system** on page 5-137
- **save_system** on page 5-143

export_hw_tcl

Description

Allows you to save the currently open system as an **_hw.tcl** file in the project directory. The saved systems appears under the **System** category in the IP Category.

Usage

```
export_hw_tcl
```

Returns

No return value.

Arguments

No arguments

Example

```
export_hw_tcl
```

get_composed_connection_parameter_value

Description

Returns the value of a parameter in a connection in a child instance containing a subsystem.

Usage

`get_composed_connection_parameter_value` *<instance> <child_connection> <parameter>*

Returns

The parameter value.

Arguments

instance

The child instance that contains a subsystem

child_connection

The name of the connection in the subsystem

parameter

The name of the parameter to query on the connection.

Example

```
get_composed_connection_parameter_value subsystem_0 cpu.data_master/memory.s0
baseAddress
```

Related Information

- get_composed_connection_parameters on page 5-91
- get_composed_connections on page 5-92

get_composed_connection_parameters

Description

Returns a list of all connections in the subsystem, for an instance that contains a subsystem.

Usage

`get_composed_connection_parameters` *<instance> <child_connection>*

Returns

A list of parameter names.

Arguments

instance

The child instance containing a subsystem.

child_connection

The name of the connection in the subsystem.

Example

```
get_composed_connection_parameters subsystem_0 cpu.data_master/memory.s0
```

Related Information

- **get_composed_connection_parameter_value** on page 5-90
- **get_composed_connections** on page 5-92

get_composed_connections

Description

For an instance that contains a subsystem of the Qsys system, returns a list of all connections found in a subsystem.

Usage

`get_composed_connections` <*instance*>

Returns

A list of connection names in the subsystem. These connection names are not qualified with the instance name.

Arguments

instance

The child instance containing a subsystem.

Example

```
get_composed_connections subsystem_0
```

Related Information

- **get_composed_connection_parameter_value** on page 5-90
- **get_composed_connection_parameters** on page 5-91

get_composed_instance_assignment

Description

For an instance that contains a subsystem of the Qsys system, returns the value of an assignment found on the instance in the subsystem.

Usage

`get_composed_instance_assignment` *<instance> <child_instance> <assignment>*

Returns

The value of the assignment.

Arguments

instance

The child instance containing a subsystem.

child_instance

The name of a child instance found in the subsystem.

assignment

The assignment key.

Example

```
get_composed_instance_assignment subsystem_0 video_0 "embeddedsw.CMacro.colorSpace"
```

Related Information

- **get_composed_instance_assignments** on page 5-94
- **get_composed_instances** on page 5-97

get_composed_instance_assignments

Description

For an instance that contains a subsystem of the Qsys system, returns a list of assignments found on the instance in the subsystem.

Usage

`get_composed_instance_assignments` *<instance> <child_instance>*

Returns

A list of assignments names.

Arguments

instance

The child instance containing a subsystem.

child_instance

The name of a child instance found in the subsystem.

Example

```
get_composed_instance_assignments subsystem_0 cpu
```

Related Information

- **get_composed_instance_assignment** on page 5-93
- **get_composed_instances** on page 5-97

get_composed_instance_parameter_value

Description

For an instance that contains a subsystem of the Qsys system, returns the value of a parameters found on the instance in the subsystem.

Usage

`get_composed_instance_parameter_value` *<instance> <child_instance> <parameter>*

Returns

The value of a parameter on the instance in the subsystem.

Arguments

instance

The child instance containing a subsystem.

child_instance

The name of a child instance found in the subsystem.

parameter

The name of the parameter to query on the instance in the subsystem.

Example

```
get_composed_instance_parameter_value subsystem_0 cpu DATA_WIDTH
```

Related Information

- **get_composed_instance_parameters** on page 5-96
- **get_composed_instances** on page 5-97

get_composed_instance_parameters

Description

For an instance that contains a subsystem of the Qsys system, returns a list of parameters found on the instance in the subsystem.

Usage

`get_composed_instance_parameters` *<instance> <child_instance>*

Returns

A list of parameters names.

Arguments

instance

The child instance containing a subsystem.

child_instance

The name of a child instance found in the subsystem.

Example

```
get_composed_instance_parameters subsystem_0 cpu
```

Related Information

- **get_composed_instance_parameter_value** on page 5-95
- **get_composed_instances** on page 5-97

get_composed_instances

Description

For an instance that contains a subsystem of the Qsys system, returns a list of child instances found in the subsystem.

Usage

`get_composed_instances` <*instance*>

Returns

A list of instance names found in the subsystem.

Arguments

instance

The child instance containing a subsystem.

Example

```
get_composed_instances subsystem_0
```

Related Information

- **get_composed_instance_assignment** on page 5-93
- **get_composed_instance_assignments** on page 5-94
- **get_composed_instance_parameter_value** on page 5-95
- **get_composed_instance_parameters** on page 5-96

get_connection_parameter_property

Description

Returns the value of a property on a parameter in a connection. Parameter properties are metadata about how Qsys uses the parameter.

Usage

`get_connection_parameter_property` *<connection> <parameter> <property>*

Returns

The value of the parameter property.

Arguments

connection

The connection to query.

parameter

The name of the parameter.

property

The property of the connection. Refer to *Parameter Properties*.

Example

```
get_connection_parameter_property cpu.data_master/dma0.csr baseAddress UNITS
```

Related Information

- **get_connection_parameter_value** on page 5-99
- **get_connection_property** on page 5-102
- **get_connections** on page 5-103
- **get_parameter_properties** on page 5-133
- **Parameter Properties** on page 5-168

get_connection_parameter_value

Description

Returns the value of a parameter on the connection. Parameters represent aspects of the connection that you can modify, such as the base address for an Avalon-MM connection.

Usage

`get_connection_parameter_value` *<connection> <parameter>*

Returns

The value of the parameter.

Arguments

connection

The connection to query.

parameter

The name of the parameter.

Example

```
get_connection_parameter_value cpu.data_master/dma0.csr baseAddress
```

Related Information

- **get_connection_parameters** on page 5-100
- **get_connections** on page 5-103
- **set_connection_parameter_value** on page 5-145

get_connection_parameters

Description

Returns a list of parameters found on a connection.

Usage

`get_connection_parameters` *<connection>*

Returns

A list of parameter names.

Arguments

connection

The connection to query.

Example

```
get_connection_parameters cpu.data_master/dma0.csr
```

Related Information

- **get_connection_parameter_property** on page 5-98
- **get_connection_parameter_value** on page 5-99
- **get_connection_property** on page 5-102

get_connection_properties

Description

Returns a list of properties found on a connection.

Usage

```
get_connection_properties
```

Returns

A list of connection properties.

Arguments

No arguments.

Example

```
get_connection_properties
```

Related Information

- get_connection_property on page 5-102
- get_connections on page 5-103

get_connection_property

Description

Returns the value of a property found on a connection. Properties represent aspects of the connection that you can modify, such as the type of connection.

Usage

`get_connection_property` *<connection> <property>*

Returns

The value of a connection property.

Arguments

connection

The connection to query.

property

The name of the connection. property. Refer to *Connection Properties*.

Example

```
get_connection_property cpu.data_master/dma0.csr TYPE
```

Related Information

- get_connection_properties on page 5-101
- Connection Properties on page 5-160

get_connections

Description

Returns a list of all connections in the system if no element is specified. If you specify a child instance, for example `cpu`, Qsys returns all connections to any interface on the instance. If specify an interface on a child instance, for example `cpu.instruction_master`, Qsys returns all connections to that interface.

Usage

`get_connections` [<*element*>]

Returns

A list of connections.

Arguments

element (optional)

The name of a child instance, or the qualified name of an interface on a child instance.

Example

```
get_connections
get_connections cpu
get_connections cpu.instruction_master
```

Related Information

- **add_connection** on page 5-80
- **remove_connection** on page 5-139

get_instance_assignment

Description

Returns the value of an assignment on a child instance. Qsys uses assignments to transfer information about hardware to embedded software tools and applications.

Usage

`get_instance_assignment` *<instance> <assignment>*

Returns

The value of the specified assignment.

Arguments

instance

The name of the child instance.

assignment

The assignment key to query.

Example

```
get_instance_assignment video_0 embeddedsw.CMacro.colorSpace
```

Related Information

get_instance_assignments on page 5-105

get_instance_assignments

Description

Returns a list of assignment keys for any assignments defined for the instance.

Usage

`get_instance_assignments` <*instance*>

Returns

A list of assignment keys.

Arguments

instance

The name of the child instance.

Example

```
get_instance_assignments sdram
```

Related Information

get_instance_assignment on page 5-104

get_instance_assignment on page 5-104

get_instance_documentation_links

Description

Returns a list of all documentation links provided by an instance.

Usage

`get_instance_documentation_links` *<instance>*

Returns

A list of documentation links.

Arguments

instance

The name of the child instance.

Example

```
get_instance_documentation_links cpu_0
```

Notes

The list of documentation links includes titles and URLs for the links. For instance, a component with a single data sheet link may return:

```
{Data Sheet} {http://url/to/data/sheet}
```

get_instance_interface_assignment

Description

Returns the value of an assignment on an interface of a child instance. Qsys uses assignments to transfer information about hardware to embedded software tools and applications.

Usage

`get_instance_interface_assignment` *<instance> <interface> <assignment>*

Returns

The value of the specified assignment.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

assignment

The assignment key to query.

Example

```
get_instance_interface_assignment sdram s1 embeddedsw.configuration.isFlash
```

Related Information

get_instance_interface_assignments on page 5-108

get_instance_interface_assignments

Description

Returns a list of assignment keys for any assignments defined for an interface of a child instance.

Usage

`get_instance_interface_assignments` *<instance> <interface>*

Returns

A list of assignment keys.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

Example

```
get_instance_interface_assignments sdram s1
```

Related Information

get_instance_interface_assignment on page 5-107

get_instance_interface_parameter_property

Description

Returns the value of a property on a parameter in an interface of a child instance. Parameter properties are metadata about how Qsys uses the parameter.

Usage

`get_instance_interface_parameter_property` *<instance> <interface> <parameter> <property>*

Returns

The value of the parameter property.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

parameter

The name of the parameter on the interface.

property

The name of the property on the parameter. Refer to *Parameter Properties*.

Example

```
get_instance_interface_parameter_property uart_0 s0 setupTime ENABLED
```

Related Information

- **get_instance_interface_parameters** on page 5-111
- **get_instance_interfaces** on page 5-116
- **get_parameter_properties** on page 5-133
- **Parameter Properties** on page 5-168

get_instance_interface_parameter_value

Description

Returns the value of a parameter of an interface in a child instance.

Usage

`get_instance_interface_parameter_value` <*instance*> <*interface*> <*parameter*>

Returns

The value of the parameter.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

parameter

The name of the parameter on the interface.

Example

```
get_instance_interface_parameter_value uart_0 s0 setupTime
```

Related Information

- **get_instance_interface_parameters** on page 5-111
- **get_instance_interfaces** on page 5-116

get_instance_interface_parameters

Description

Returns a list of parameters for an interface in a child instance.

Usage

`get_instance_interface_parameters` *<instance> <interface>*

Returns

A list of parameter names for parameters in the interface.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the uart_0 s0.

Example

```
get_instance_interface_parameters instance interface
```

Related Information

- **get_instance_interface_parameter_value** on page 5-110
- **get_instance_interfaces** on page 5-116

get_instance_interface_port_property

Description

Returns the value of a property of a port found in the interface of a child instance.

Usage

`get_instance_interface_port_property` *<instance> <interface> <port> <property>*

Returns

The value of the port property.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

port

The name of the port in the interface.

property

The name of the property of the port. Refer to *Port Properties*.

Example

```
get_instance_interface_port_property uart_0 exports tx WIDTH
```

Related Information

- **get_instance_interface_ports** on page 5-113
- **get_port_properties** on page 5-134
- **Port Properties** on page 5-173

get_instance_interface_ports

Description

Returns a list of ports found in an interface of a child instance.

Usage

`get_instance_interface_ports` *<instance> <interface>*

Returns

A list of port names found in the interface.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

Example

```
get_instance_interface_ports uart_0 s0
```

Related Information

- **get_instance_interface_port_property** on page 5-112
- **get_instance_interfaces** on page 5-116

get_instance_interface_properties

Description

Returns a list of properties that can be queried for an interface in a child instance.

Usage

```
get_instance_interface_properties
```

Returns

A list of property names.

Arguments

No arguments.

Example

```
get_instance_interface_properties
```

Related Information

- get_instance_interface_property on page 5-115
- get_instance_interfaces on page 5-116

get_instance_interface_property

Description

Returns the value of a property for an interface in a child instance.

Usage

`get_instance_interface_property` *<instance> <interface> <property>*

Returns

The value of the property.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

property

The name of the property of the interface. Refer to *Element Properties*.

Example

```
get_instance_interface_property uart_0 s0 DESCRIPTION
```

Related Information

- get_instance_interface_properties on page 5-114
- get_instance_interfaces on page 5-116
- Element Properties on page 5-163

get_instance_interfaces

Description

Returns a list of interfaces found in a child instance

Usage

`get_instance_interfaces` <*instance*>

Returns

A list of interface names.

Arguments

instance

The name of the child instance.

Example

```
get_instance_interfaces uart_0
```

Related Information

- **get_instance_interface_ports** on page 5-113
- **get_instance_interface_properties** on page 5-114
- **get_instance_interface_property** on page 5-115

get_instance_parameter_property

Description

Returns the value of a property on a parameter in a child instance. Parameter properties are metadata about how Qsys uses the parameter.

Usage

`get_instance_parameter_property` *<instance> <parameter> <property>*

Returns

The value of the parameter property.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter in the instance.

property

The name of the property of the parameter. Refer to *Parameter Properties*.

Example

```
get_instance_parameter_property uart_0 baudRate ENABLED
```

Related Information

- **get_instance_parameters** on page 5-119
- **get_parameter_properties** on page 5-133
- **Parameter Properties** on page 5-168

get_instance_parameter_value

Description

Returns the value of a parameter in a child instance.

Usage

`get_instance_parameter_value` *<instance> <parameter>*

Returns

The value of the parameter.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter in the instance.

Example

```
get_instance_parameter_value uart_0 baudRate
```

Related Information

- **get_instance_parameters** on page 5-119
- **set_instance_parameter_value** on page 5-146

get_instance_parameters

Description

Returns a list of parameters in a child instance.

Usage

`get_instance_parameters` *<instance>*

Returns

A list of parameters in the instance.

Arguments

instance

The name of the child instance.

Example

```
get_instance_parameters uart_0
```

Related Information

- **get_instance_parameter_property** on page 5-117
- **get_instance_interface_parameter_value** on page 5-110
- **set_instance_parameter_value** on page 5-146

get_instance_port_property

Description

Returns the value of a property of a port contained by an interface in a child instance.

Usage

`get_instance_port_property` *<instance> <port> <property>*

Returns

The value of the property for the port.

Arguments

instance

The name of the child instance.

port

The name of a port in one of the interfaces on the child instance.

property

The name of a property found on the port. Refer to *Port Properties*.

Example

```
get_instance_port_property uart_0 tx WIDTH
```

Related Information

- **get_instance_interface_ports** on page 5-113
- **get_port_properties** on page 5-134
- **Port Properties** on page 5-173

get_instance_properties

Description

Returns a list of properties for a child instance.

Usage

```
get_instance_properties
```

Returns

A list of property names for the child instance.

Arguments

No arguments.

Example

```
get_instance_properties
```

Related Information

get_instance_property on page 5-122

get_instance_property

Description

Returns the value of a property for a child instance.

Usage

`get_instance_property` *<instance> <property>*

Returns

The value of the property.

Arguments

instance

The name of the child instance.

property

The name of a property found on the instance. Refer to *Element Properties*.

Example

```
get_instance_property uart_0 ENABLED
```

Related Information

- **get_instance_properties** on page 5-121
- **Element Properties** on page 5-163

get_instances

Description

Returns a list of the instance names for all child instances in the system.

Usage

```
get_instances
```

Returns

A list of child instance names.

Arguments

No arguments.

Example

```
get_instances
```

Related Information

- **add_instance** on page 5-81
- **remove_instance** on page 5-141

get_interconnect_requirement

Description

Returns the value of an interconnect requirement for a system or interface on a child instance.

Usage

`get_interconnect_requirement` *<element_id> <requirement>*

Returns

The value of the interconnect requirement.

Arguments

element_id

`{$system}` for the system, or the qualified name of the interface of an instance, in `<instance>.<interface>` format. In Tcl, the system identifier is escaped, for example, `{$system}`.

requirement

The name of the requirement.

Example

```
get_interconnect_requirement {$system} qsys_mm.maxAdditionalLatency
```

get_interconnect_requirements

Description

Returns a list of all interconnect requirements in the system.

Usage

```
get_interconnect_requirements
```

Returns

A flattened list of interconnect requirements. Every sequence of three elements in the list corresponds to one interconnect requirement. The first element in the sequence is the element identifier. The second element is the requirement name. The third element is the value. You can loop over the returned list with a `foreach loop`, for example:

```
foreach { element_id name value } $requirement_list { loop_body
         }
```

Arguments

No arguments.

Example

```
get_interconnect_requirements
```

get_interface_port_property

Description

Returns the value of a property of a port contained by one of the top-level exported interfaces

Usage

`get_interface_port_property` *<interface> <port> <property>*

Returns

The value of the property.

Arguments

interface

The name of a top-level interface on the system.

port

The name of a port found in the interface.

property

The name of a property found on the port. Refer to *Port Properties*.

Example

```
get_interface_port_property uart_exports tx DIRECTION
```

Related Information

- **get_interface_ports** on page 5-127
- **get_port_properties** on page 5-134
- **Port Properties** on page 5-173

get_interface_ports

Description

Returns the names of all of the ports that have been added to a given interface.

Usage

`get_interface_ports` *<interface>*

Returns

A list of port names.

Arguments

interface

The name of a top-level interface on the system.

Example

```
get_interface_ports export_clk_out
```

Related Information

- **get_interface_port_property** on page 5-126
- **get_interfaces** on page 5-130

get_interface_properties

Description

Returns the names of all the available interface properties common to all interface types.

Usage

```
get_interface_properties
```

Returns

A list of interface properties.

Arguments

No arguments.

Example

```
get_interface_properties
```

Related Information

- **get_interface_property** on page 5-129
- **set_interface_property** on page 5-149

get_interface_property

Description

Returns the value of a single interface property from the specified interface.

Usage

`get_interface_property` *<interface> <property>*

Returns

The property value.

Arguments

interface

The name of a top-level interface on the system.

property

The name of the property. Refer to *Interface Properties*.

Example

```
get_interface_property export_clk_out EXPORT_OF
```

Related Information

- **get_interface_properties** on page 5-128
- **set_interface_property** on page 5-149
- **Interface Properties** on page 5-165

get_interfaces

Description

Returns a list of top-level interfaces in the system.

Usage

```
get_interfaces
```

Returns

A list of the top-level interfaces exported from the system.

Arguments

No arguments.

Example

```
get_interfaces
```

Related Information

- **add_interface** on page 5-82
- **get_interface_ports** on page 5-127
- **get_interface_property** on page 5-129
- **remove_interface** on page 5-142
- **set_interface_property** on page 5-149

get_module_properties

Description

Returns the properties that you can manage for top-level module of the Qsys system.

Usage

```
get_module_properties
```

Returns

A list of property names.

Arguments

No arguments.

Example

```
get_module_properties
```

Related Information

- **get_module_property** on page 5-132
- **set_module_property** on page 5-150

get_module_property

Description

Returns the value of a top-level system property.

Usage

`get_module_property` *<property>*

Returns

The value of the property.

Arguments

property

The name of the property to query. Refer to *Module Properties*.

Example

```
get_module_property NAME
```

Related Information

- **get_module_properties** on page 5-131
- **set_module_property** on page 5-150

get_parameter_properties

Description

Returns a list of properties that you can query for any parameters, for example parameters on instances, interfaces, instance interfaces, and connections.

Usage

```
get_parameter_properties
```

Returns

A list of parameter properties.

Arguments

No arguments.

Example

```
get_parameter_properties
```

Related Information

- **get_connection_parameter_property** on page 5-98
- **get_instance_interface_parameter_property** on page 5-109
- **get_instance_parameter_property** on page 5-117

get_port_properties

Description

Returns a list of properties that you can query for ports.

Usage

```
get_port_properties
```

Returns

A list of port properties.

Arguments

No arguments.

Example

```
get_port_properties
```

Related Information

- **get_instance_interface_port_property** on page 5-112
- **get_instance_interface_ports** on page 5-113
- **get_instance_port_property** on page 5-120
- **get_interface_port_property** on page 5-126
- **get_interface_ports** on page 5-127

get_project_properties

Description

Returns a list of properties that you can query for properties pertaining to the Quartus II project.

Usage

```
get_project_properties
```

Returns

A list of project properties.

Arguments

No arguments

Example

```
get_project_properties
```

Related Information

- get_project_property on page 5-136
- set_project_property on page 5-151

get_project_property

Description

Returns the value of a Quartus II project property. Not all Quartus II project properties are available.

Usage

`get_project_property` *<property>*

Returns

The value of the property.

Arguments

property

The name of the project property. Refer to *Project properties*.

Example

```
get_project_property DEVICE_FAMILY
```

Related Information

- get_module_properties on page 5-131
- get_module_property on page 5-132
- set_module_property on page 5-150

load_system

Description

Loads a Qsys system from a file, and uses the system as the current system for scripting commands.

Usage

`load_system` *<file>*

Returns

No return value.

Arguments

file

The path to a **.qsys** file.

Example

```
load_system example.qsys
```

Related Information

- **create_system** on page 5-88
- **save_system** on page 5-143

lock_avalon_base_address

Description

Prevents the memory-mapped base address from being changed for connections to the specified interface on an instance when Qsys runs the `auto_assign_base_addresses` or `auto_assign_system_base_addresses` commands.

Usage

`lock_avalon_base_address` *<instance.interface>*

Returns

No return value.

Arguments

instance.interface

The qualified name of the interface of an instance, in <*`instance`*>.<*`interface`*> format.

Example

```
lock_avalon_base_address sdram.s1
```

Related Information

- auto_assign_base_addresses on page 5-84
- auto_assign_system_base_addresses on page 5-85
- unlock_avalon_base_address on page 5-154

remove_connection

Description

This command removes a connection from the system.

Usage

`remove_connection` *<connection>*

Returns

no return value

Arguments

connection

The name of the connection to remove

Example

```
remove_connection cpu.data_master/sdram.s0
```

Related Information

- **add_connection** on page 5-80
- **get_connections** on page 5-103

remove_dangling_connections

Description

Removes connections where both end points of the connection no longer exist in the system.

Usage

```
remove_dangling_connections
```

Returns

No return value.

Arguments

No arguments.

Example

```
remove_dangling_connections
```

remove_instance

Description

Removes a child instance from the system.

Usage

`remove_instance` <*instance*>

Returns

No return value.

Arguments

instance

The name of the child instance to remove.

Example

```
remove_instance cpu
```

Related Information

- **add_instance** on page 5-81
- **get_instances** on page 5-123

remove_interface

Description

Removes an exported top-level interface from the system.

Usage

`remove_interface` *<interface>*

Returns

No return value.

Arguments

interface

The name of the exported top-level interface.

Example

```
remove_interface clk_out
```

Related Information

- **add_interface** on page 5-82
- **get_interfaces** on page 5-130

save_system

Description

Saves the current system to the named file. If you do not specify the file, Qsys saves the system to the same file that was opened with the `load_system` command. You can specify the file as an absolute or relative path. Relative paths are relative to directory of the most recently loaded system, or relative to the working directory if no systems are loaded.

Usage

`save_system` <*file*>

Returns

No return value.

Arguments

file

If available, the path of the **.qsys** file to save.

Example

```
save_system

save_system file.qsys
```

send_message

Description

Sends a message to the user of the script. The message text is normally interpreted as HTML. You can use the <*b*> element to provide emphasis.

Usage

`send_message` <*level*> <*message*>

Returns

No return value.

Arguments

level

The following message levels are supported:

- `ERROR`--Provides an error message.
- `WARNING`--Provides a warning message.
- `INFO`--Provides an informational message.
- `PROGRESS`--Provides a progress message.
- `DEBUG`--Provides a debug message when debug mode is enabled. Refer to *Message Levels Properties*.

message

The text of the message.

Example

```
send_message ERROR "The system is down!"
```

Related Information

Message Levels Properties on page 5-166

set_connection_parameter_value

Description

Sets the value of a parameter for a connection.

Usage

`set_connection_parameter_value` *<connection> <parameter> <value>*

Returns

No return value.

Arguments

connection

The name if the connection.

parameter

The name of the parameter.

value

The new parameter value.

Example

```
set_connection_parameter_value cpu.data_master/dma0.csr baseAddress "0x000a0000"
```

Related Information

- get_connection_parameter_value on page 5-99
- get_connection_parameters on page 5-100

set_instance_parameter_value

Description

Set the value of a parameter for a child instance. You cannot set derived parameters and `SYSTEM_INFO` parameters for the child instance with this command.

Usage

`set_instance_parameter_value` *<instance> <parameter> <value>*

Returns

No return value.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter.

value

The new parameter value.

Example

```
set_instance_parameter_value uart_0 baudRate 9600
```

Related Information

- **get_instance_parameter_value** on page 5-118
- **get_instance_parameter_property** on page 5-117

set_instance_property

Description

Sets the value of a property of a child instance. Most instance properties are read-only and can only be set by the instance itself. The primary use for this command is to update the `ENABLED` parameter, which includes or excludes a child instance when generating Qsys interconnect.

Usage

`set_instance_property` *<instance> <property> <value>*

Returns

No return value.

Arguments

instance

The name of the child instance.

property

The name of the property. Refer to *Instance Properties*.

value

The new property value.

Example

```
set_instance_property cpu ENABLED false
```

Related Information

- **get_instance_parameters** on page 5-119
- **get_instance_property** on page 5-122
- **Instance Properties** on page 5-164

set_interconnect_requirement

Description

Sets the value of an interconnect requirement for a system or an interface on a child instance.

Usage

`set_interconnect_requirement` *<element_id> <requirement> <value>*

Returns

No return value.

Arguments

element_id

`{$system}` for the system, or qualified name of the interface of an instance, in `<`*`instance`*`>.<`*`interface`*`>` format. In Tcl, the system identifier is escaped, for example, `{$system}`.

requirement

The name of the requirement.

value

The new requirement value.

Example

```
set_interconnect_requirement {$system} qsys_mm.clockCrossingAdapter HANDSHAKE
```

set_interface_property

Description

Sets the value of a property on an exported top-level interface. You use this command to set the `EXPORT_OF` property to specify which interface of a child instance is exported via this top-level interface.

Usage

`set_interface_property` *<interface> <property> <value>*

Returns

No return value.

Arguments

interface

The name of an exported top-level interface.

property

The name of the property. Refer to *Interface Properties*.

value

The new property value.

Example

```
set_interface_property clk_out EXPORT_OF clk.clk_out
```

Related Information

- **add_interface** on page 5-82
- **get_interface_properties** on page 5-128
- **get_interface_property** on page 5-129
- **Interface Properties** on page 5-165

set_module_property

Description

Sets the value of a system property, such as the name of the system using the `NAME` property.

Usage

`set_module_property` *<property> <value>*

Returns

No return value.

Arguments

property

The name of the property. Refer to *Module Properties*.

value

The new property value.

Example

```
set_module_property NAME "new_system_name"
```

Related Information

- **get_module_properties** on page 5-131
- **get_module_property** on page 5-132
- **Module Properties** on page 5-167

set_project_property

Description

Sets the value of a project property, such as the device family.

Usage

`set_project_property` *<property> <value>*

Returns

No return value.

Arguments

property

The name of the property. Refer to *Project Properties*.

value

The new property value.

Example

```
set_project_property DEVICE_FAMILY "Cyclone IV GX"
```

Related Information

- **get_project_properties** on page 5-135
- **set_project_property** on page 5-151
- **Project Properties** on page 5-174
- **get_project_properties** on page 5-135
- **get_project_property** on page 5-136
- **Project Properties** on page 5-174

set_use_testbench_naming_pattern

Description

Use this command to create testbench systems so that the generated file names for the test system match the system's original generated file names. Without setting this, the generated file names for the test system receive the top-level testbench system name.

Usage

`set_use_testbench_naming_pattern` *<value>*

Returns

No return value.

Arguments

value

True or false.

Example

```
set_use_testbench_naming_pattern true
```

Notes

Use this command only to create testbench systems.

set_validation_property

Description

Sets a property that affects how and when validation is run. To disable system validation after each scripting command, set `AUTOMATIC_VALIDATION` to `False`.

Usage

`set_validation_property` *<property>* *<value>*

Returns

No return value.

Arguments

property

The name of the property. Refer to *Validation Properties*.

value

The new property value.

Example

```
set_validation_property AUTOMATIC_VALIDATION false
```

Related Information

- **validate_system** on page 5-158
- **Validation Properties** on page 5-178

unlock_avalon_base_address

Description

Allows the memory-mapped base address to change for connections to the specified interface on an instance when Qsys runs the `auto_assign_base_addresses` or `auto_assign_system_base_addresses` commands.

Usage

`unlock_avalon_base_address` <*instance.interface*>

Returns

No return value.

Arguments

instance.interface

The qualified name of the interface of an instance, in <*`instance`*>.<*`interface`*> format.

Example

```
unlock_avalon_base_address sdram.s1
```

Related Information

- **auto_assign_base_addresses** on page 5-84
- **auto_assign_system_base_addresses** on page 5-85
- **lock_avalon_base_address** on page 5-138

validate_connection

Description

Validates the specified connection and returns validation messages.

Usage

`validate_connection` <*connection*>

Returns

A list of messages produced during validation.

Arguments

connection

The name of the connection to validate.

Example

```
validate_connection cpu.data_master/sdram.s1
```

Related Information

- **validate_instance** on page 5-156
- **validate_instance_interface** on page 5-157
- **validate_system** on page 5-158

validate_instance

Description

Validates the specified child instance and returns validation messages.

Usage

`validate_instance` *<instance>*

Returns

A list of messages produced during validation.

Arguments

instance

The name of the child instance to validate.

Example

```
validate_instance cpu
```

Related Information

- **validate_connection** on page 5-155
- **validate_instance_interface** on page 5-157
- **validate_system** on page 5-158

validate_instance_interface

Description

Validates an interface on a child instance and returns validation messages.

Usage

`validate_instance_interface` *<instance> <interface>*

Returns

A list of messages produced during validation.

Arguments

instance

The name of a child instance.

interface

The name of the interface on the child instance to validate.

Example

```
validate_instance_interface cpu data_master
```

Related Information

- **validate_connection** on page 5-155
- **validate_instance** on page 5-156
- **validate_system** on page 5-158

validate_system

Description

Validates the system and returns validation messages.

Usage

```
validate_system
```

Returns

A list of validation messages produced during validation.

Arguments

No arguments.

Example

```
validate_system
```

Related Information

- validate_connection on page 5-155
- validate_instance on page 5-156
- validate_instance_interface on page 5-157

Qsys Scripting Property Reference

Interface properties work differently for **_hw.tcl** scripting than with qsys scripting. In **_hw.tcl**, interfaces do not distinguish between properties and parameters. In qsys scripting, properties and parameters are unique.

Connection Properties

Type	Name	Description
string	`END`	The end interface of the connection.
string	`NAME`	The name of the connection.
string	`START`	The start interface of the connection.
String	`TYPE`	The type of the connection.

Design Environment Type Properties

Description

IP cores use the design environment to identify what sort of interfaces are most appropriate to connect in the parent system.

Name	Description
`NATIVE`	The design environment supports native IP interfaces.
`QSYS`	The design environment supports standard Qsys interfaces.

Direction Properties

Name	Description
`BIDIR`	The direction for a bidirectional signal.
`INOUT`	The direction for an input signal.
`OUTPUT`	The direction for an output signal.

Element Properties

Description

Element properties are, with the exception of ENABLED and NAME, read-only properties of the types of instances, interfaces, and connections. These read-only properties represent metadata that does not vary between copies of the same type. ENABLED and NAME properties are specific to particular instances, interfaces, or connections.

Type	Name	Description
String	AUTHOR	The author of the component or interface.
Boolean	AUTO_EXPORT	Indicates whether unconnected interfaces on the instance are automatically exported.
String	CLASS_NAME	The type of the instance, interface or connection, for example, altera_nios2 or avalon_slave.
String	DESCRIPTION	The description of the instance, interface or connection type.
String	DISPLAY_NAME	The display name for referencing the type of instance, interface or connection.
Boolean	EDITABLE	Indicates whether you can edit the component in the Qsys Component Editor.
Boolean	ENABLED	Indicates whether the instance is turned on.
String	GROUP	The IP Catalog category.
Boolean	INTERNAL	Hides internal IP components or sub-components from the IP Catalog..
String	NAME	The name of the instance, interface or connection.
String	VERSION	The version number of the instance, interface or connection, for example, 14.0.

Instance Properties

Type	Name	Description
String	`AUTO_EXPORT`	Indicates whether unconnected interfaces on the instance are automatically exported.
Boolean	`ENABLED`	If true, this instance is included in the generated system. if false, it is not included.
String	`NAME`	The name of the system, which is used as the name of the top-level module in the generated HDL.

Interface Properties

Type	Name	Description
String	`EXPORT_OF`	Indicates which interface of a child instance to export through the top-level interface. Before using this command, you must create the top-level interface using the `add_interface` command. You must use the format: `<`*`instanceName.interfaceName`*`>`. For example: `set_interface_property CSC_input EXPORT_OF my_colorSpace-Converter.input_port`

Message Levels Properties

Name	Description
`COMPONENT_INFO`	Reports an informational message only during component editing.
`DEBUG`	Provides messages when debug mode is turned on.
`ERROR`	Provides an error message.
`INFO`	Provides an informational message.
`PROGRESS`	Reports progress during generation.
`TODOERROR`	Provides an error message that indicates the system is incomplete.
`WARNING`	Provides a warning message.

Module Properties

Type	Name	Description
String	`GENERATION_ID`	The generation ID for the system.
String	`NAME`	The name of the instance.

Parameter Properties

Type	Name	Description
Boolean	`AFFECTS_ELABORATION`	Set `AFFECTS_ELABORATION` to false for parameters that do not affect the external interface of the module. An example of a parameter that does not affect the external interface is `isNonVolatileStorage`. An example of a parameter that does affect the external interface is `width`. When the value of a parameter changes and `AFFECTS_ELABORATION` is false, the elaboration phase does not repeat and improves performance. When `AFFECTS_ELABORATION` is set to true, the default value, Qsys reanalyzes the HDL file to determine the port widths and configuration each time a parameter changes.
Boolean	`AFFECTS_GENERATION`	The default value of `AFFECTS_GENERATION` is false if you provide a top-level HDL module. The default value is true if you provide a fileset callback. Set `AFFECTS_GENERATION` to false if the value of a parameter does not change the results of fileset generation.
Boolean	`AFFECTS_VALIDATION`	The `AFFECTS_VALIDATION` property determines whether a parameter's value sets derived parameters, and whether the value affects validation messages. When set to false, this may improve response time in the parameter editor when the value changes.
String[]	`ALLOWED_RANGES`	Indicates the range or ranges of the parameter. For integers, Each range is a single value, or a range of values defined by a start and end value, and delimited by a colon, for example, `11:15`. This property also specifies the legal values and description strings for integers, for example, `{0:None 1:Monophonic 2:Stereo 4:Quadrophonic}`, where 0, 1, 2, and 4 are the legal values. You can assign description strings in the parameter editor for string variables. For example, `ALLOWED_RANGES {"dev1:Cyclone IV GX""dev2:Stratix V GT"}`
String	`DEFAULT_VALUE`	The default value.
Boolean	`DERIVED`	When `True`, indicates that the parameter value is set by the component and cannot be set by the user. Derived parameters are not saved as part of an instance's parameter values. The default value is `False`.
String	`DESCRIPTION`	A short user-visible description of the parameter, suitable for a tooltip description in the parameter editor.
String[]	`DISPLAY_HINT`	Provides a hint about how to display a property.

Type	Name	Description
		• boolean--For integer parameters whose value are 0 or 1. The parameter displays as an option that you can turn on or off. • radio--Displays a parameter with a list of values as radio buttons. • hexadecimal--For integer parameters, displays and interprets the value as a hexadecimal number, for example: 0x00000010 instead of 16. • fixed_size--For string_list and integer_list parameters, the fixed_size DISPLAY_HINT eliminates the **Add** and **Remove** buttons from tables.
String	DISPLAY_NAME	The GUI label that appears to the left of this parameter.
String	DISPLAY_UNITS	The GUI label that appears to the right of the parameter.
Boolean	ENABLED	When False, the parameter is turned off. It displays in the parameter editor but is grayed out, indicating that you cannot edit this parameter.
String	GROUP	Controls the layout of parameters in the GUI.
Boolean	HDL_PARAMETER	When True, Qsys passes the parameter to the HDL component description. The default value is False.
String	LONG_DESCRIPTION	A user-visible description of the parameter. Similar to DESCRIPTION, but allows a more detailed explanation.
String	NEW_INSTANCE_VALUE	Changes the default value of a parameter without affecting older components that do not explicitly set a parameter value, and use the DEFAULT_VALUE property. Oder instances continue to use DEFAULT_VALUE for the parameter and new instances use the value assigned by NEW_INSTANCE_VALUE.
String[]	SYSTEM_INFO	Allows you to assign information about the instantiating system to a parameter that you define. SYSTEM_INFO requires an argument specifying the type of information for example, `SYSTEM_INFO <info-type>`
String	SYSTEM_INFO_ARG	Defines an argument to pass to SYSTEM_INFO. For example, the name of a reset interface.
(various)	SYSTEM_INFO_TYPE	Specifies the types of system information that you can query. Refer to *System Info Type Properties*.
(various)	TYPE	Specifies the type of the parameter. Refer to *Parameter Type Properties*.
(various)	UNITS	Sets the units of the parameter. Refer to *Units Properties*.
Boolean	VISIBLE	Indicates whether or not to display the parameter in the parameter editor.

Type	Name	Description
String	`WIDTH`	Indicates the width of the logic vector for the `STD_LOGIC_VECTOR` parameter.

Related Information

- **System Info Type Properties** on page 5-175
- **Parameter Type Properties** on page 5-172
- **Units Properties** on page 5-177

Parameter Status Properties

Type	Name	Description
Boolean	`ACTIVE`	Indicates that this parameter is an active parameter.
Boolean	`DEPRECATED`	Indicates that this parameter exists only for backwards compatibility, and may not have any effect.
Boolean	`EXPERIMENTAL`	Indicates that this parameter is experimental and not exposed in the design flow.

Parameter Type Properties

Name	Description
BOOLEAN	A boolean parameter set to true or false.
FLOAT	A signed 32-bit floating point parameter. (Not supported for HDL parameters.)
INTEGER	A signed 32-bit integer parameter.
INTEGER_LIST	A parameter that contains a list of 32-bit integers. (Not supported for HDL parameters.)
LONG	A signed 64-bit integer parameter. (Not supported for HDL parameters.)
NATURAL	A 32-bit number that contains values 0 to 2147483647 (0x7fffffff).
POSITIVE	A 32-bit number that contains values 1 to 2147483647 (0x7fffffff).
STD_LOGIC	A single bit parameter set to 0 or 1.
STD_LOGIC_VECTOR	An arbitrary-width number. The parameter property WIDTH determines the size of the logic vector.
STRING	A string parameter.
STRING_LIST	A parameter that contains a list of strings. (Not supported for HDL parameters.)

Port Properties

Type	Name	Description
(various)	`DIRECTION`	The direction of the signal. Refer to *Direction Properties*.
String	`ROLE`	The type of the signal. Each interface type defines a set of interface types for its ports.
Integer	`WIDTH`	The width of the signal in bits.

Project Properties

Type	Name	Description
String	`DEVICE`	The device part number in the Quartus II project that contains the Qsys system.
String	`DEVICE_FAMILY`	The device family name in the Quartus II project that contains the Qsys system.

System Info Type Properties

Type	Name	Description
String	`ADDRESS_MAP`	An XML-formatted string that describes the address map for the interface specified in the `SYSTEM_INFO` parameter property.
Integer	`ADDRESS_WIDTH`	The number of address bits that Qsys requires to address memory-mapped slaves connected to the specified memory-mapped master in this instance.
String	`AVALON_SPEC`	The version of the Qsys interconnect. Refer to *Avalon Interface Specifications*.
Integer	`CLOCK_DOMAIN`	An integer that represents the clock domain for the interface specified in the `SYSTEM_INFO` parameter property. If this instance has interfaces on multiple clock domains, you can use this property to determine which interfaces are on each clock domain. The absolute value of the integer is arbitrary.
Long, Integer	`CLOCK_RATE`	The rate of the clock connected to the clock input specified in the `SYSTEM_INFO` parameter property. If zero, the clock rate is currently unknown.
String	`CLOCK_RESET_INFO`	The name of this instance's primary clock or reset sink interface. You use this property to determine the reset sink for global reset when you use SOPC Builder interconnect that conforms to *Avalon Interface Specifications*.
String	`CUSTOM_INSTRUCTION_SLAVES`	Provides slave information, including the name, base address, address span, and clock cycle type.
String	`DESIGN_ENVIRONMENT`	A string that identifies the current design environment. Refer to *Design Environment Type Properties*.
String	`DEVICE`	The device part number of the selected device.
String	`DEVICE_FAMILY`	The family name of the selected device.
String	`DEVICE_FEATURES`	A list of key/value pairs delimited by spaces that indicate whether a device feature is available in the selected device family. The format of the list is suitable for passing to the `array` command. The keys are device features. The values are 1 if the feature is present, and 0 if the feature is absent.
String	`DEVICE_SPEEDGRADE`	The speed grade of the selected device.
Integer	`GENERATION_ID`	A integer that stores a hash of the generation time that Qsys uses as a unique ID for a generation run.

Type	Name	Description
BigInteger, Long	`INTERRUPTS_USED`	A mask indicating which bits of an interrupt receiver are connected to interrupt senders. The interrupt receiver is specified in the system info argument.
Integer	`MAX_SLAVE_DATA_WIDTH`	The data width of the widest slave connected to the specified memory-mapped master.
String, Boolean, Integer	`QUARTUS_INI`	The value of the **quartus.ini** setting specified in the system info argument.
Integer	`RESET_DOMAIN`	An integer representing the reset domain for the interface specified in the `SYSTEM_INFO` parameter property If this instance has interfaces on multiple reset domains, you can use this property to determine which interfaces are on each reset domain. The absolute value of the integer is arbitrary.
String	`TRISTATECONDUIT_INFO`	An XML description of the tri-state conduit masters connected to a tri-state conduit slave. The slave is specified as the `SYSTEM_INFO` parameter property. The value contains information about the slave, connected master instance and interface names, and signal names, directions, and widths.
String	`TRISTATECONDUIT_MASTERS`	The names of the instance's interfaces that are tri-state conduit slaves.
String	`UNIQUE_ID`	A string guaranteed to be unique to this instance.

Related Information

- Design Environment Type Properties on page 5-161
- Avalon Interface Specifications
- Qsys Interconnect on page 7-1

Units Properties

Name	Description
`ADDRESS`	A memory-mapped address.
`BITS`	Memory size in bits.
`BITSPERSECOND`	Rate in bits per second.
`BYTES`	Memory size in bytes.
`CYCLES`	A latency or count in clock cycles.
`GIGABITSPERSECOND`	Rate in gigabits per second.
`GIGABYTES`	Memory size in gigabytes.
`GIGAHERTZ`	Frequency in GHz.
`HERTZ`	Frequency in Hz.
`KILOBITSPERSECOND`	Rate in kilobits per second.
`KILOBYTES`	Memory size in kilobytes.
`KILOHERTZ`	Frequency in kHz.
`MEGABITSPERSECOND`	Rate, in megabits per second.
`MEGABYTES`	Memory size in megabytes.
`MEGAHERTZ`	Frequency in MHz.
`MICROSECONDS`	Time in microseconds.
`MILLISECONDS`	Time in milliseconds.
`NANOSECONDS`	Time in nanoseconds.
`NONE`	Unspecified units.
`PERCENT`	A percentage.
`PICOSECONDS`	Time in picoseconds.
`SECONDS`	Time in seconds.

Validation Properties

Type	Name	Description
Boolean	`AUTOMATIC_VALIDATION`	When `true`, Qsys runs system validation and elaboration after each scripting command. When `false`, Qsys runs system validation with validation scripting commands. Some queries affected by system elaboration may be incorrect if automatic validation is turned off. You can disable validation to make a system script run faster.

Document Revision History

The table below indicates edits made to the *Creating a System With Qsys* content since its creation.

Table 5-16: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• New figure: *Avalon-MM Write Master Timing Waveforms in the Parameters Tab.* • Added **Enable ECC protection** option, *Specify Qsys Interconnect Requirements.* • Added External Memory Interface Debug Toolkit note, *Generate a Qsys System.* • Modelsim-Altera now supports native mixed-language (VHDL/Verilog/SystemVerilog) simulation, *Generating Files for Synthesis and Simulation.*
December 2014	14.1.0	• Create and Manage Hierarchical Qsys Systems. • Schematic tab. • View and Filter Clock and Reset Domains. • **File** > **Recent Projects** menu item. • Updated example: Hierarchical System Using Instance Parameters
August 2014	14.0a10.0	• Added distinction between legacy and standard device generation. • Updated: *Upgrading Outdated IP Components.* • Updated: *Generating a Qsys System.* • Updated: *Integrating a Qsys System with the Quartus II Software.* • Added screen shot: *Displaying Your Qsys System.*
June 2014	14.0.0	• Added tab descriptions: Details, Connections. • Added *Managing IP Settings in the Quartus II Software.* • Added *Upgrading Outdated IP Components.* • Added *Support for Avalon-MM Non-Power of Two Data Widths.*
November 2013	13.1.0	• Added *Integrating with the .qsys File.* • Added *Using the Hierarchy Tab.* • Added *Managing Interconnect Requirements.* • Added *Viewing Qsys Interconnect.*

Date	Version	Changes
May 2013	13.0.0	• Added AMBA APB support. • Added qsys-generate utility. • Added VHDL BFM ID support. • Added *Creating Secure Systems (TrustZones)* . • Added *CMSIS Support for Qsys Systems With An HPS Component*. • Added VHDL language support options.
November 2012	12.1.0	• Added AMBA AXI4 support.
June 2012	12.0.0	• Added AMBA AX3I support. • Added Preset Editor updates. • Added command-line utilities, and scripts.
November 2011	11.1.0	• Added Synopsys VCS and VCS MX Simulation Shell Script. • Added Cadence Incisive Enterprise (NCSIM) Simulation Shell Script. • Added *Using Instance Parameters and Example Hierarchical System Using Parameters*.
May 2011	11.0.0	• Added simulation support in Verilog HDL and VHDL. • Added testbench generation support. • Updated simulation and file generation sections.
December 2010	10.1.0	Initial release.

Related Information

Quartus II Handbook Archive

Creating Qsys Components 6

2015.05.04

QII5V1 Subscribe Send Feedback

In order to describe and package IP components for use in a Qsys system, you must create a Hardware Component Definition File (**_hw.tcl**) which will describes your component, its interfaces and HDL files. Qsys provides the Component Editor to help you create a simple **_hw.tcl** file.

The **Demo AXI Memory** example on the **Qsys Design Examples** page of the Altera web site provides the full code examples that appear in the following topics.

Qsys supports Avalon, AMBA AXI3 (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB3 (version 1.0) interface specifications.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specifications
- Demo AXI Memory Example

Qsys Components

A Qsys component includes the following elements:

- Information about the component type, such as name, version, and author.
- HDL description of the component's hardware, including SystemVerilog, Verilog HDL, or VHDL files
- Constraint files (Synopsys Design Constraints File (**.sdc**) and/or Quartus II IP File (**.qip**)) that define the component for synthesis and simulation.
- A component's interfaces, including I/O signals.
- The parameters that configure the operation of the component.

IP Component Interface Support in Qsys

IP components can have any number of interfaces in any combination. Each interface represents a set of signals that you can connect within a Qsys system, or export outside of a Qsys system.

Qsys IP components can include the following interface types:

Table 6-1: IP Component Interface Types

Interface Type	Description
Memory-Mapped	Connects memory-referencing master devices with slave memory devices. Master devices may be processors and DMAs, while slave memory devices may be RAMs, ROMs, and control registers. Data transfers between master and slave may be uni-directional (read only or write only), or bi-directional (read and write).
Streaming	Connects Avalon Streaming (Avalon-ST) sources and sinks that stream unidirectional data, as well as high-bandwidth, low-latency IP components. Streaming creates datapaths for unidirectional traffic, including multichannel streams, packets, and DSP data. The Avalon-ST interconnect is flexible and can implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet, Interlaken, and video. You can define bus widths, packets, and error conditions.
Interrupts	Connects interrupt senders to interrupt receivers. Qsys supports individual, single-bit interrupt requests (IRQs). In the event that multiple senders assert their IRQs simultaneously, the receiver logic (typically under software control) determines which IRQ has highest priority, then responds appropriately
Clocks	Connects clock output interfaces with clock input interfaces. Clock outputs can fan-out without the use of a bridge. A bridge is required only when a clock from an external (exported) source connects internally to more than one source.
Resets	Connects reset sources with reset input interfaces. If your system requires a particular positive-edge or negative-edge synchronized reset, Qsys inserts a reset controller to create the appropriate reset signal. If you design a system with multiple reset inputs, the reset controller ORs all reset inputs and generates a single reset output.
Conduits	Connects point-to-point conduit interfaces, or represent signals that are exported from the Qsys system. Qsys uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Qsys system as a point-to-point connection, or conduit interfaces can be exported and brought to the top-level of the system as top-level system I/O. You can use conduits to connect to external devices, for example external DDR SDRAM memory, and to FPGA logic defined outside of the Qsys system.

Component Structure

Altera provides components automatically installed with the Quartus II software. You can obtain a list of Qsys-compliant components provided by third-party IP developers on Altera's **Intellectual Property & Reference Designs** page by typing: **qsys certified** in the **Search** box, and then selecting **IP Core & Reference Designs**. Components are also provided with Altera development kits, which are listed on the **All Development Kits** page.

Every component is defined with a < *component_name* >**_hw.tcl file**, a text file written in the Tcl scripting language that describes the component to Qsys. When you design your own custom component, you can create the **_hw.tcl file** manually, or by using the Qsys Component Editor.

The Component Editor simplifies the process of creating **_hw.tcl** files by creating a file that you can edit outside of the Component Editor to add advanced procedures. When you edit a previously saved **_hw.tcl** file, Qsys automatically backs up the earlier version as **_hw.tcl~**.

You can move component files into a new directory, such as a network location, so that other users can use the component in their systems. The **_hw.tcl** file contains relative paths to the other files, so if you move an **_hw.tcl** file, you should also move all the HDL and other files associated with it.

There are three component types:

- **Static**— Static components always generate the same output, regardless of their parameterization. Components that instantiate static components must have only static children.
- **Generated**—A generated component's fileset callback allows an instance of the component to create unique HDL design files based on the instance's parameter values.
- **Composed**—Composed components are subsystems constructed from instances of other components. You can use a composition callback to manage the subsystem in a composed component.

Related Information

- **Create a Composed Component or Subsystem** on page 6-28
- **Add Component Instances to a Static or Generated Component** on page 6-31
- **Intellectual Property & Reference Designs**

Component File Organization

A typical component uses the following directory structure where the names of the directories are not significant:

<component_directory>/

- **<hdl>/**—Contains the component HDL design files, for example **.v**, **.sv**, or **.vhd** files that contain the top-level module, along with any required constraint files.
- **<component_name> _hw.tcl**—The component description file.
- **<component_name> _sw.tcl**—The software driver configuration file. This file specifies the paths for the **.c** and **.h** files associated with the component, when required.
- **<software>/**—Contains software drivers or libraries related to the component.

Note: Refer to the *Nios II Software Developer's Handbook* for information about writing a device driver or software package suitable for use with the Nios II processor.

Related Information

- **Hardware Abstraction LayerTool Reference (Nios II Software Developer's Handbook)**
- **Nios II Software Build Tool Reference (Nios II Software Developer's Handbook)**

Component Versions

Qsys systems support multiple versions of the same component within the same system; you can create and maintain multiple versions of the same component.

If you have multiple **_hw.tcl** files for components with the same NAME module properties and different VERSION module properties, both versions of the component are available.

If multiple versions of the component are available in the IP Catalog, you can add a specific version of a component by right-clicking the component, and then selecting **Add version** <*version_number*>.

Upgrade IP Components to the Latest Version

When you open a Qsys design, if Qsys detects IP components that require regeneration, the **Upgrade IP Cores** dialog box appears and allows you to upgrade outdated components.

Components that you must upgrade in order to successfully compile your design appear in red. Status icons indicate whether a component is currently being regenerated, the component is encrypted, or that there is not enough information to determine the status of component. To upgrade a component, in the **Upgrade IP Cores** dialog box, select the component that you want to upgrade, and then click **Upgrade**. The Quartus II software maintains a list of all IP components associated with your design on the **Components** tab in the Project Navigator.

Related Information

Upgrade IP Components Dialog Box

Design Phases of an IP Component

When you define a component with the Qsys Component Editor, or a custom **_hw.tcl** file, you specify the information that Qsys requires to instantiate the component in a Qsys system and to generate the appropriate output files for synthesis and simulation.

The following phases describe the process when working with components in Qsys:

- **Discovery**—During the discovery phase, Qsys reads the **_hw.tcl** file to identify information that appears in the IP Catalog, such as the component's name, version, and documentation URLs. Each time you open Qsys, the tool searches for the following file types using the default search locations and entries in the **IP Search Path**:
  - **_hw.tcl** files—Each **_hw.tcl** file defines a single component.
  - IP Index (**.ipx**) files—Each **.ipx** file indexes a collection of available components, or a reference to other directories to search.
- **Static Component Definition**—During the static component definition phase, Qsys reads the **_hw.tcl** file to identify static parameter declarations, interface properties, interface signals, and HDL files that define the component. At this stage of the life cycle, the component interfaces may be only partially defined.
- **Parameterization**—During the parameterization phase, after an instance of the component is added to a Qsys system, the user of the component specifies parameters with the component's parameter editor.
- **Validation**—During the validation phase, Qsys validates the values of each instance's parameters against the allowed ranges specified for each parameter. You can use callback procedures that run during the validation phase to provide validation messages. For example, if there are dependencies between parameters where only certain combinations of values are supported, you can report errors for the unsupported values.

- **Elaboration**—During the elaboration phase, Qsys queries the component for its interface information. Elaboration is triggered when an instance of a component is added to a system, when its parameters are changed, or when a system property changes. You can use callback procedures that run during the elaboration phase to dynamically control interfaces, signals, and HDL files based on the values of parameters. For example, interfaces defined with static declarations can be enabled or disabled during elaboration. When elaboration is complete, the component's interfaces and design logic must be completely defined.
- **Composition**—During the composition phase, a component can manipulate the instances in the component's subsystem. The **_hw.tcl** file uses a callback procedure to provide parameterization and connectivity of sub-components.
- **Generation**—During the generation phase, Qsys generates synthesis or simulation files for each component in the system into the appropriate output directories, as well as any additional files that support associated tools.

Create IP Components in the Qsys Component Editor

The Qsys Component Editor, accessed by clicking **New Component** in the IP Catalog, allows you to create and package a component for use in Qsys. When you use the Component Editor to define a component, the Component Editor writes the information to an **_hw.tcl** file.

The Component Editor allows you to perform the following tasks:

- Specify component’s identifying information, such as name, version, author, etc.
- Specify the SystemVerilog, Verilog HDL, or VHDL files, and constraint files that define the component for synthesis and simulation.
- Create an HDL template for a component by first defining its parameters, signals, and interfaces.
- Associate and define signals for a component’s interfaces.
- Set parameters on interfaces, which specify characteristics.
- Specify relationships between interfaces.
- Declare parameters that alter the component structure or functionality.

If the component is HDL-based, you must define the parameters and signals in the HDL file, and cannot add or remove them in the Component Editor. If you have not yet created the top-level HDL file, you declare the parameters and signals in the Component Editor, and they are then included in the HDL template file that Qsys creates.

In a Qsys system, the interfaces of a component are connected within the system, or exported as top-level signals from the system.

If you are creating the component using an existing HDL file, the order in which the tabs appear in the Component Editor reflects the recommended design flow for component development. You can use the **Prev** and **Next** buttons at the bottom of the Component Editor window to guide you through the tabs.

If the component is not based on an existing HDL file, enter the parameters, signals, and interfaces first, and then return to the **Files** tab to create the top-level HDL file template. When you click **Finish**, Qsys creates the component **_hw.tcl** file with the details provided on the Component Editor tabs.

After the component is saved, it is available in the IP Catalog.

If you require features in the component that are not supported by the Component Editor, such as callback procedures, you can use the Component Editor to create the **_hw.tcl** file, and then manually edit the file to complete the component definition. Subsequent topics document the **_hw.tcl** commands that are generated by the Component Editor, as well as some of the advanced features that you can add with your own **_hw.tcl** commands.

Note: By default, custom components do not have registered outputs, even if they are exported out of the Qsys system. For a custom component, if you want to export the signals, you must add the registered outputs.

Example 6-1: Qsys Creates an _hw.tcl File from Entries in the Component Editor

```
#
# connection point clock
#
add_interface clock clock end
set_interface_property clock clockRate 0
set_interface_property clock ENABLED true

add_interface_port clock clk clk Input 1

#
# connection point reset
#
add_interface reset reset end
set_interface_property reset associatedClock clock
set_interface_property reset synchronousEdges DEASSERT
set_interface_property reset ENABLED true

add_interface_port reset reset_n reset_n Input 1

#
# connection point streaming
#
add_interface streaming avalon_streaming start
set_interface_property streaming associatedClock clock
set_interface_property streaming associatedReset reset
set_interface_property streaming dataBitsPerSymbol 8
set_interface_property streaming errorDescriptor ""
set_interface_property streaming firstSymbolInHighOrderBits true
set_interface_property streaming maxChannel 0
set_interface_property streaming readyLatency 0
set_interface_property streaming ENABLED true

add_interface_port streaming aso_data data Output 8
add_interface_port streaming aso_valid valid Output 1
add_interface_port streaming aso_ready ready Input 1

#
# connection point slave
#
add_interface slave axi end
set_interface_property slave associatedClock clock
set_interface_property slave associatedReset reset
set_interface_property slave readAcceptanceCapability 1
set_interface_property slave writeAcceptanceCapability 1
set_interface_property slave combinedAcceptanceCapability 1
set_interface_property slave readDataReorderingDepth 1
set_interface_property slave ENABLED true

add_interface_port slave axs_awid awid Input AXI_ID_W
...
add_interface_port slave axs_rresp rresp Output 2
```

Related Information

- Component Interface Tcl Reference on page 9-1

Save an IP Component and Create the _hw.tcl File

You save a component by clicking **Finish** in the Qsys Component Editor. The Component Editor saves the component as <*component_name*> **_hw.tcl** file.

Altera recommends that you save **_hw.tcl** files and their associated files in an **ip/** <*class-name*> directory within your Quartus II project directory. You can also create components to use with other applications, such as the C compiler and a board support package (BSP) generator.

Refer to *Creating a System with Qsys* for information on how to search for and add components to the IP Catalog for use in your designs.

Related Information

Publishing Component Information to Embedded Software (Nios II Software Developer's Handbook)

Creating a System with Qsys on page 5-1

Edit an IP Component with the Qsys Component Editor

In Qsys, you make changes to a component by right-clicking the component in the **System Contents** tab, and then clicking **Edit**. After making changes, click **Finish** to save the changes to the **_hw.tcl** file.

You can open an **_hw.tcl** file in a text editor to view the hardware Tcl for the component. If you edit the **_hw.tcl** file to customize the component with advanced features, you cannot use the Component Editor to make further changes without over-writing your customized file.

You cannot use the Component Editor to edit components installed with the Quartus II software, such as Altera-provided components. If you edit the HDL for a component and change the interface to the top-level module, you must edit the component to reflect the changes you make to the HDL.

Related Information

Creating Qsys Components

Specify IP Component Type Information

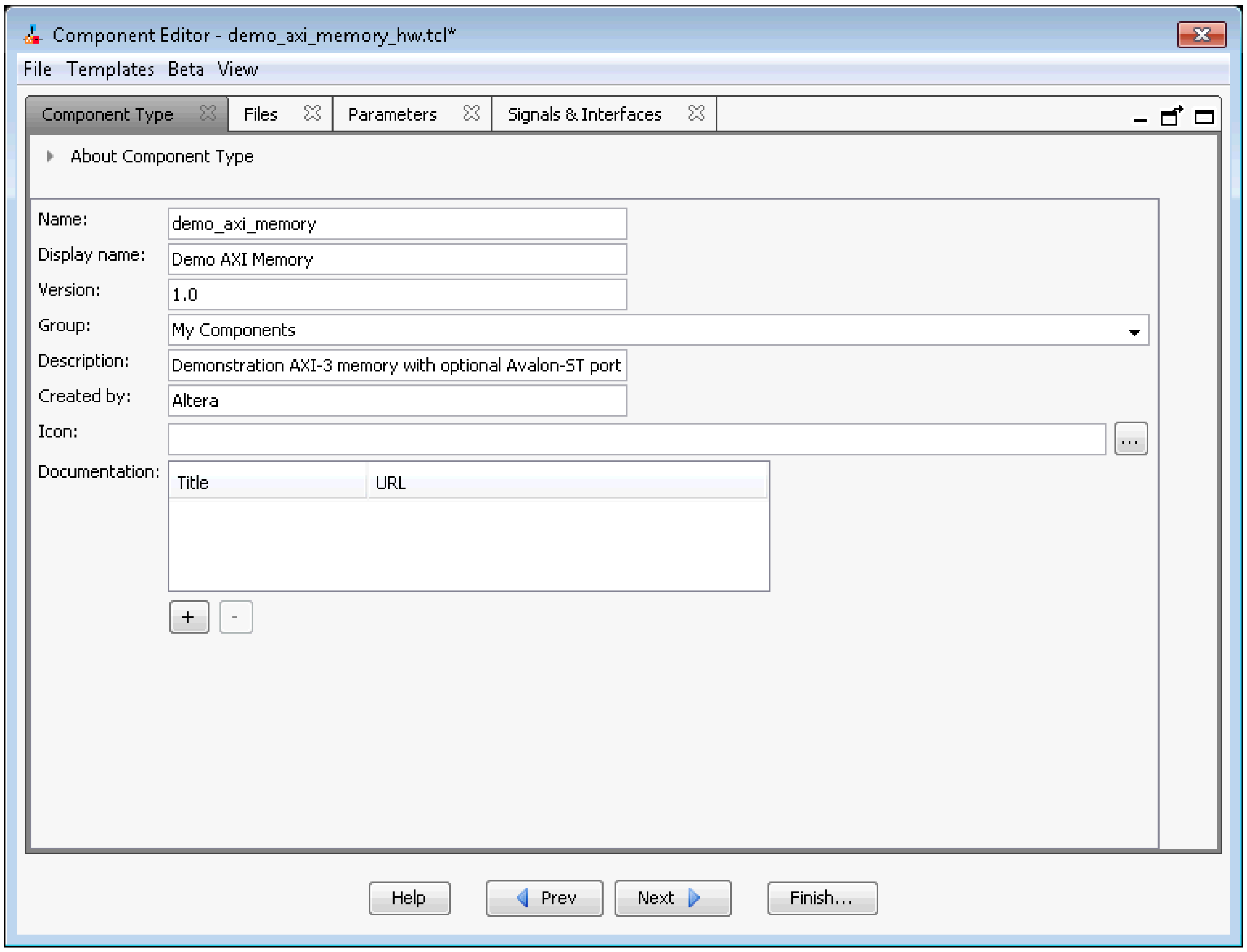
The **Component Type** tab in the Qsys Component Editor allows you to specify the following information about the component:

- **Name**—Specifies the name used in the **_hw.tcl** filename, as well as in the top-level module name when you create a synthesis wrapper file for a non HDL-based component.
- **Display name**—Identifies the component in the parameter editor, which you use to configure and instance of the component, and also appears in the IP Catalog under **Project** and on the **System Contents** tab.
- **Version**—Specifies the version number of the component.
- **Group**—Represents the category of the component in the list of available components in the IP Catalog. You can select an existing group from the list, or define a new group by typing a name in the **Group** box. Separating entries in the **Group** box with a slash defines a subcategory. For example, if you type **Memories and Memory Controllers/On-Chip**, the component appears in the IP Catalog under the **On-Chip** group, which is a subcategory of the **Memories and Memory Controllers** group. If you save the component in the project directory, the component appears in the IP Catalog in the group you specified under **Project**. Alternatively, if you save the component in the Quartus II installation directory, the component appears in the specified group under **IP Catalog**.
- **Description**—Allows you to describe the component. This description appears when the user views the component details.
- **Created By**—Allows you to specify the author of the component.
- **Icon**—Allows you to enter the relative path to an icon file (**.gif**, **.jpg**, or **.png** format) that represents the component and appears as the header in the parameter editor for the component. The default image is the Altera MegaCore function icon.
- **Documentation**—Allows you to add links to documentation for the component, and appears when you right-click the component in the IP Catalog, and then select **Details**.

  - To specify an Internet file, begin your path with **http://**, for example: **http://mydomain.com/datasheets/my_memory_controller.html**.
  - To specify a file in the file system, begin your path with **file:///** for Linux, and **file:////** for Windows; for example (Windows): **file:////company_server/datasheets my_memory_controller.pdf**.

Figure 6-1: Component Type Tab in the Component Editor

The **Display name**, **Group**, **Description**, **Created By**, **Icon**, and **Documentation** entries are optional.



Name: demo_axi_memory
Display name: Demo AXI Memory
Version: 1.0
Group: My Components
Description: Demonstration AXI-3 memory with optional Avalon-ST port
Created by: Altera
Icon:
Documentation:

Title	URL

When you use the Component Editor to create a component, it writes this basic component information in the **_hw.tcl file**. The example below shows the component hardware Tcl code related to the entries for the **Component Type** tab in figure above. The `package require` command specifies the Quartus II software version that Qsys uses to create the **_hw.tcl** file, and ensures compatibility with this version of the Qsys API in future ACDS releases.

Example 6-2: _hw.tcl Created from Entries in the Component Type Tab

The component defines its basic information with various module properties using the `set_module_property` command. For example, `set_module_property NAME` specifies the name of the component, while `set_module_property VERSION` allows you to specify the version of the component. When you apply a version to the **_hw.tcl** file, it allows the file to behave exactly the same way in future releases of the Quartus II software.

```
# request TCL package from ACDS 14.0

package require -exact qsys 14.0

# demo_axi_memory
```

```
set_module_property DESCRIPTION \
"Demo AXI-3 memory with optional Avalon-ST port"

set_module_property NAME demo_axi_memory
set_module_property VERSION 1.0
set_module_property GROUP "My Components"
set_module_property AUTHOR Altera
set_module_property DISPLAY_NAME "Demo AXI Memory"
```

Related Information

- Component Interface Tcl Reference on page 9-1

Create an HDL File in the Qsys Component Editor

If you do not have an HDL file for your component, you can use the Qsys Component Editor to define the component signals, interfaces, and parameters of your component, and then create a simple top-level HDL file.

You can then edit the HDL file to add the logic that describes the component's behavior.

1. In the Qsys Component Editor, specify the information about the component in the **Signals & Interfaces**, and **Interfaces**, and **Parameters** tabs.
2. Click the **Files** tab.
3. Click **Create Synthesis File from Signals**.
   The Component Editor creates an HDL file from the specified signals, interfaces, and parameters, and the **.v** file appears in the **Synthesis File** table.

Related Information

Specify Synthesis and Simulation Files in the Qsys Component Editor on page 6-12

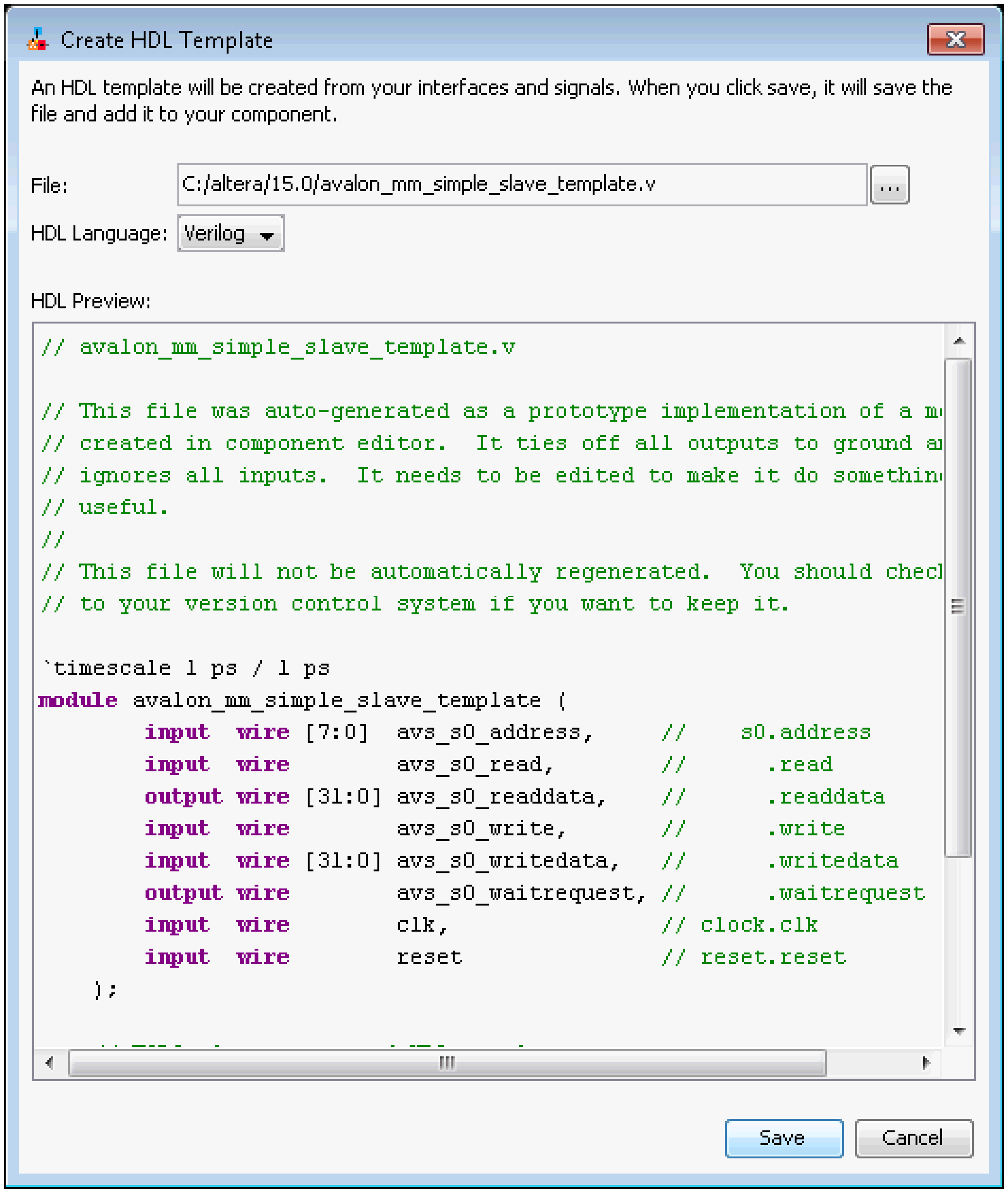
Create an HDL File Using a Template in the Qsys Component Editor

You can use a template to create interfaces and signals for your Qsys component

1. In Qsys, click **new_component** in the IP Catalog.
2. On the **Component Type** tab, define your component information in the **Name**, **Display Name**, **Version**, **Group**, **Description**, **Created by**, **Icon**, and **Documentation** boxes.
3. Click **Finish**.
   Your new component appears in the IP Catalog under the category that you define for "Group".
4. In Qsys, right-click your new component in the IP Catalog, and then click **Edit**.
5. In the Qsys Component Editor, click any interface from the Templates drop-down menu.
   The Component Editor fills the **Signals** and **Interfaces** tabs with the component interface template details.
6. On the **Files** tab, click **Create Synthesis File from Signals**.
7. Do the following in the **Create HDL Template** dialog box as shown below:

a. Verify that the correct files appears in **File** path, or browse to the location where you want to save your file.
b. Select the HDL language.
c. Click **Save** to save your new interface, or **Cancel** to discard the new interface definition.

Create HDL Template Dialog Box



An HDL template will be created from your interfaces and signals. When you click save, it will save the file and add it to your component.

File: C:/altera/15.0/avalon_mm_simple_slave_template.v

HDL Language: Verilog

HDL Preview:

```
// avalon_mm_simple_slave_template.v

// This file was auto-generated as a prototype implementation of a mo
// created in component editor.  It ties off all outputs to ground an
// ignores all inputs.  It needs to be edited to make it do somethin
// useful.
//
// This file will not be automatically regenerated.  You should chec
// to your version control system if you want to keep it.

`timescale 1 ps / 1 ps
module avalon_mm_simple_slave_template (
        input  wire [7:0]  avs_s0_address,     //    s0.address
        input  wire        avs_s0_read,        //      .read
        output wire [31:0] avs_s0_readdata,    //      .readdata
        input  wire        avs_s0_write,       //      .write
        input  wire [31:0] avs_s0_writedata,   //      .writedata
        output wire        avs_s0_waitrequest, //      .waitrequest
        input  wire        clk,                // clock.clk
        input  wire        reset               // reset.reset
    );
```

8. Verify the **<component_name>.v** file appears in the **Synthesis Files** table on the **Files** tab.

Related Information

Specify Synthesis and Simulation Files in the Qsys Component Editor on page 6-12

Specify Synthesis and Simulation Files in the Qsys Component Editor

The **Files** tab in the Qsys Component Editor allows you to specify synthesis and simulation files for your custom component.

If you already an HDL files that describe the behavior and structure of your component, you can specify those files on the **Files** tab.

If you do not yet have an HDL file, you can specify the signals, interfaces, and parameters of the component in the Component Editor, and then use the **Create Synthesis File from Signals** option on the **Files** tab to create the top-level HDL file. The Component Editor generates the **_hw.tcl** commands to specify the files.

Note: After you analyze the component's top-level HDL file (on the **Files** tab), you cannot add or remove signals or change the signal names on the **Signals & Interfaces** tab. If you need to edit signals, edit your HDL source, and then click **Create Synthesis File from Signals** on the **Files** tab to integrate your changes.

A component uses filesets to specify the different sets of files that you can generate for an instance of the component. The supported fileset types are: `QUARTUS_SYNTH`, for synthesis and compilation in the Quartus II software, `SIM_VERILOG`, for Verilog HDL simulation, and `SIM_VHDL`, for VHDL simulation.

In an **_hw.tcl** file, you can add a fileset with the `add_fileset` command. You can then list specific files with the `add_fileset_file` command. The `add_fileset_property` command allows you to add properties such as `TOP_LEVEL`.

You can populate a fileset with a a fixed list of files, add different files based on a parameter value, or even generate an HDL file with a custom HDL generator function outside of the **_hw.tcl** file.

Related Information

- Create an HDL File in the Qsys Component Editor on page 6-10
- Create an HDL File Using a Template in the Qsys Component Editor on page 6-10

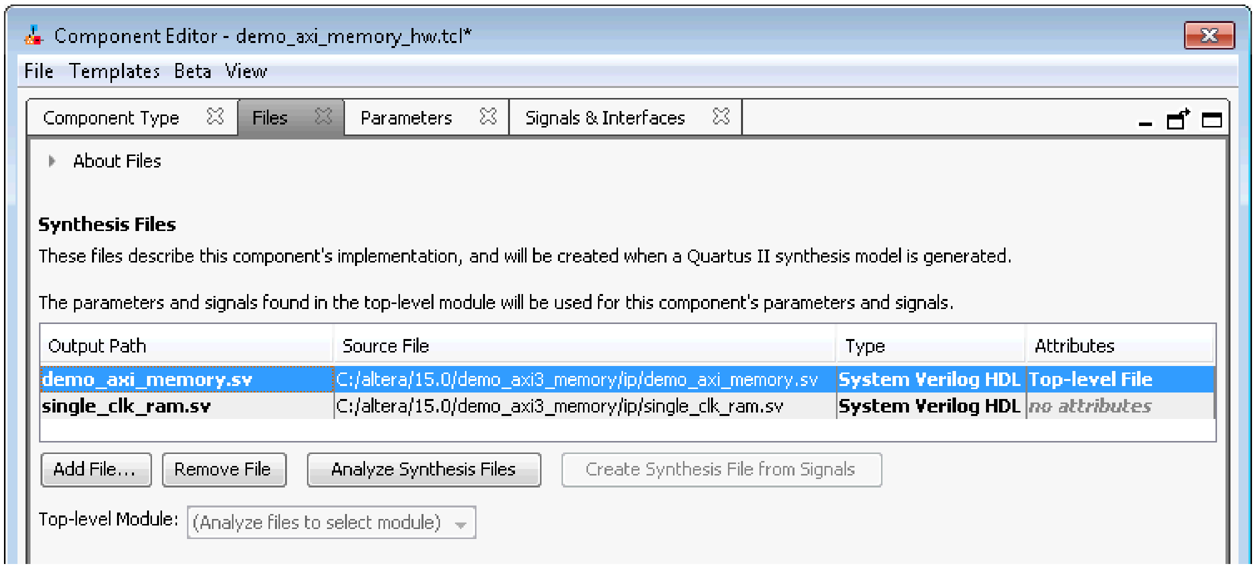
Specify HDL Files for Synthesis in the Qsys Component Editor

In the Qsys Component Editor, you can add HDL files and other support files with options on the **Files** tab.

A component must specify an HDL file as the top-level file. The top-level HDL file contains the top-level module. The **Synthesis Files** list may also include supporting HDL files, such as timing constraints, or other files required to successfully synthesize and compile in the Quartus II software. The synthesis files for a component are copied to the generation output directory during Qsys system generation.

Figure 6-2: Using HDL Files to Define a Component

In the **Synthesis Files** section on the **Files** tab in the Qsys Component Editor, the **demo_axi_memory.sv** file should be selected as the top-level file for the component.



Component Editor - demo_axi_memory_hw.tcl*

File Templates Beta View

Component Type | Files | Parameters | Signals & Interfaces

About Files

Synthesis Files

These files describe this component's implementation, and will be created when a Quartus II synthesis model is generated.

The parameters and signals found in the top-level module will be used for this component's parameters and signals.

Output Path	Source File	Type	Attributes
demo_axi_memory.sv	C:/altera/15.0/demo_axi3_memory/ip/demo_axi_memory.sv	System Verilog HDL	Top-level File
single_clk_ram.sv	C:/altera/15.0/demo_axi3_memory/ip/single_clk_ram.sv	System Verilog HDL	no attributes

Top-level Module: (Analyze files to select module)

Analyze Synthesis Files in the Qsys Component Editor

After you specify the top-level HDL file in the Qsys Component Editor, click **Analyze Synthesis Files** to analyze the parameters and signals in the top-level, and then select the top-level module from the **Top Level Module** list. If there is a single module or entity in the HDL file, Qsys automatically populates the **Top-level Module** list.

Once analysis is complete and the top-level module is selected, you can view the parameters and signals on the **Parameters** and **Signals & Interfaces** tabs. The Component Editor may report errors or warnings at this stage, because the signals and interfaces are not yet fully defined.

Note: At this stage in the Component Editor flow, you cannot add or remove parameters or signals created from a specified HDL file without editing the HDL file itself.

The synthesis files are added to a fileset with the name `QUARTUS_SYNTH` and type `QUARTUS_SYNTH` in the **_hw.tcl** file created by the Component Editor. The top-level module is used to specify the `TOP_LEVEL` fileset property. Each synthesis file is individually added to the fileset. If the source files are saved in a different directory from the working directory where the **_hw.tcl** is located, you can use standard fixed or relative path notation to identify the file location for the `PATH` variable.

Example 6-3: _hw.tcl Created from Entries in the Files tab in the Synthesis Files Section

```
# file sets

add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL demo_axi_memory

add_fileset_file demo_axi_memory.sv
SYSTEM_VERILOG PATH demo_axi_memory.sv
```

```
add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v
```

Related Information

Specify HDL Files for Synthesis in the Qsys Component Editor on page 6-12

Component Interface Tcl Reference on page 9-1

Name HDL Signals for Automatic Interface and Type Recognition in the Qsys Component Editor

If you create the component's top-level HDL file before using the Component Editor, the Component Editor recognizes the interface and signal types based on the signal names in the source HDL file. This auto-recognition feature eliminates the task of manually assigning each interface and signal type in the Component Editor.

To enable auto-recognition, you must create signal names using the following naming convention:

<interface type prefix>_<interface name>_<signal type>

Specifying an interface name with *<interface name>* is optional if you have only one interface of each type in the component definition. For interfaces with only one signal, such as clock and reset inputs, the *<interface type prefix>* is also optional.

Table 6-2: Interface Type Prefixes for Automatic Signal Recognition

When the Component Editor recognizes a valid prefix and signal type for a signal, it automatically assigns an interface and signal type to the signal based on the naming convention. If no interface name is specified for a signal, you can choose an interface name on the **Signals & Interfaces** tab in the Component Editor.

Interface Prefix	Interface Type
asi	Avalon-ST sink (input)
aso	Avalon-ST source (output)
avm	Avalon-MM master
avs	Avalon-MM slave
axm	AXI master
axs	AXI slave
apm	APB master
aps	APB slave
coe	Conduit
csi	Clock Sink (input)

Interface Prefix	Interface Type
cso	Clock Source (output)
inr	Interrupt receiver
ins	Interrupt sender
ncm	Nios II custom instruction master
ncs	Nios II custom instruction slave
rsi	Reset sink (input)
rso	Reset source (output)
tcm	Avalon-TC master
tcs	Avalon-TC slave

Refer to the *Avalon Interface Specifications* or the *AMBA Protocol Specification* for the signal types available for each interface type.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specification

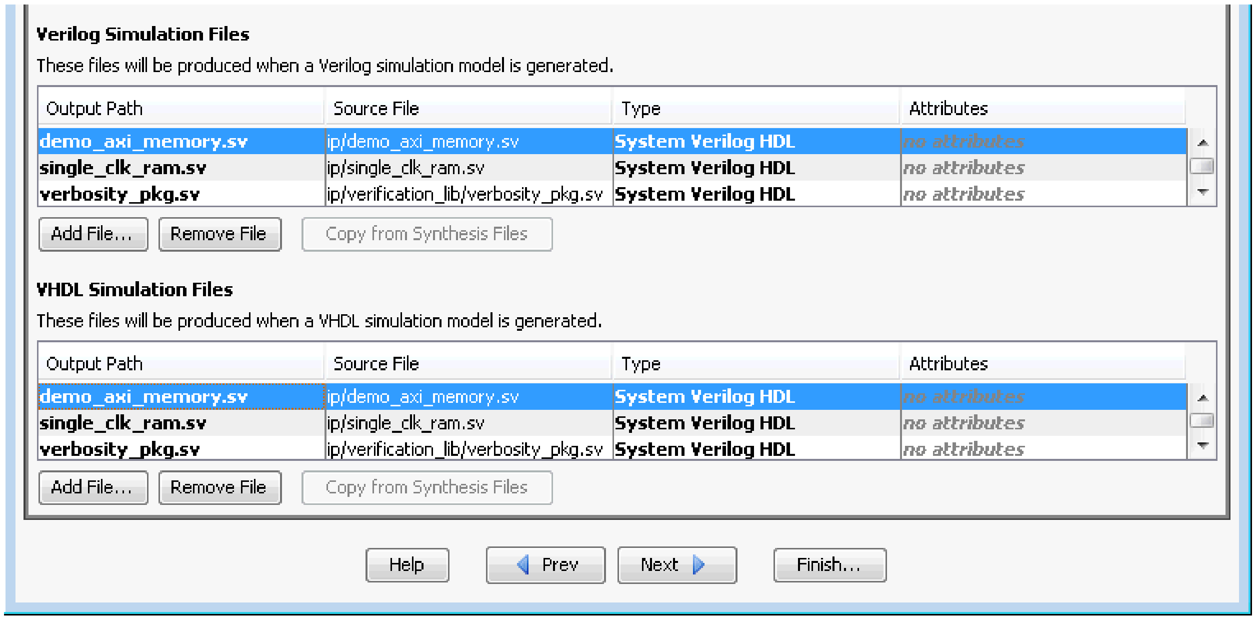
Specify Files for Simulation in the Component Editor

To support Qsys system generation for your custom component, you must specify VHDL or Verilog simulation files.

You can choose to generate Verilog or VHDL simulation files. In most cases, these files are the same as the synthesis files. If there are simulation-specific HDL files or simulation models, you can use them in addition to, or in place of the synthesis files. To use your synthesis files as your simulation files, click **Copy From Synthesis Files** on the **Files** tab in the Qsys Component Editor.

Note: The order that you add files to the fileset determines the order of compilation. For VHDL filesets with VHDL files, you must add the files bottom-up, adding the top-level file last.

Figure 6-3: Specifying the Simulation Output Files on the Files Tab



Verilog Simulation Files

These files will be produced when a Verilog simulation model is generated.

Output Path	Source File	Type	Attributes
demo_axi_memory.sv	ip/demo_axi_memory.sv	System Verilog HDL	no attributes
single_clk_ram.sv	ip/single_clk_ram.sv	System Verilog HDL	no attributes
verbosity_pkg.sv	ip/verification_lib/verbosity_pkg.sv	System Verilog HDL	no attributes

VHDL Simulation Files

These files will be produced when a VHDL simulation model is generated.

Output Path	Source File	Type	Attributes
demo_axi_memory.sv	ip/demo_axi_memory.sv	System Verilog HDL	no attributes
single_clk_ram.sv	ip/single_clk_ram.sv	System Verilog HDL	no attributes
verbosity_pkg.sv	ip/verification_lib/verbosity_pkg.sv	System Verilog HDL	no attributes

You specify the simulation files in a similar way as the synthesis files with the fileset commands in a **_hw.tcl** file. The code example below shows `SIM_VERILOG` and `SIM_VHDL` filesets for Verilog and VHDL simulation output files. In this example, the same Verilog files are used for both Verilog and VHDL outputs, and there is one additional System Verilog file added. This method works for designers of Verilog IP to support users who want to generate a VHDL top-level simulation file when they have a mixed-language simulation tool and license that can read the Verilog output for the component.

Example 6-4: _hw.tcl Created from Entries in the Files tab in the Simulation Files Section

```
add_fileset SIM_VERILOG SIM_VERILOG "" ""
set_fileset_property SIM_VERILOG TOP_LEVEL demo_axi_memory
add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv

add_fileset_file demo_axi_memory.sv SYSTEM_VERILOG PATH \
demo_axi_memory.sv

add_fileset SIM_VHDL SIM_VHDL "" ""
set_fileset_property SIM_VHDL TOP_LEVEL demo_axi_memory
set_fileset_property SIM_VHDL ENABLE_RELATIVE_INCLUDE_PATHS false

add_fileset_file demo_axi_memory.sv SYSTEM_VERILOG PATH \
demo_axi_memory.sv

add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv
```

Related Information

- Component Interface Tcl Reference on page 9-1

Include an Internal Register Map Description in the .svd for Slave Interfaces Connected to an HPS Component

Qsys supports the ability for IP component designers to specify register map information on their slave interfaces. This allows components with slave interfaces that are connected to an HPS component to include their internal register description in the generated **.svd** file.

To specify their internal register map, the IP component designer must write and generate their own **.svd** file and attach it to the slave interface using the following command:

`set_interface_property` <*slave interface*> `CMSIS_SVD_FILE` <*file path*>

The `CMSIS_SVD_VARIABLES` interface property allows for variable substitution inside the **.svd** file. You can dynamically modify the character data of the **.svd** file by using the CMSIS_SVD_VARIABLES property.

Example 6-5: Setting the CMSIS_SVD_VARIBLES Interface Property

For example, if you set the `CMSIS_SVD_VARIABLES` in the **_hw tcl** file, then in the **.svd** file if there is a variable `{width}` that describes the element `<size>${width}</size>`, it is replaced by `<size>23</size>` during generation of the **.svd** file. Note that substitution works only within character data (the data enclosed by <element>...</element>) and not on element attributes.

```
set_interface_property <interface name> \
CMSIS_SVD_VARIABLES "{width} {23}"
```

Related Information

Component Interface Tcl Reference on page 9-1

CMSIS - Cortex Microcontroller Software

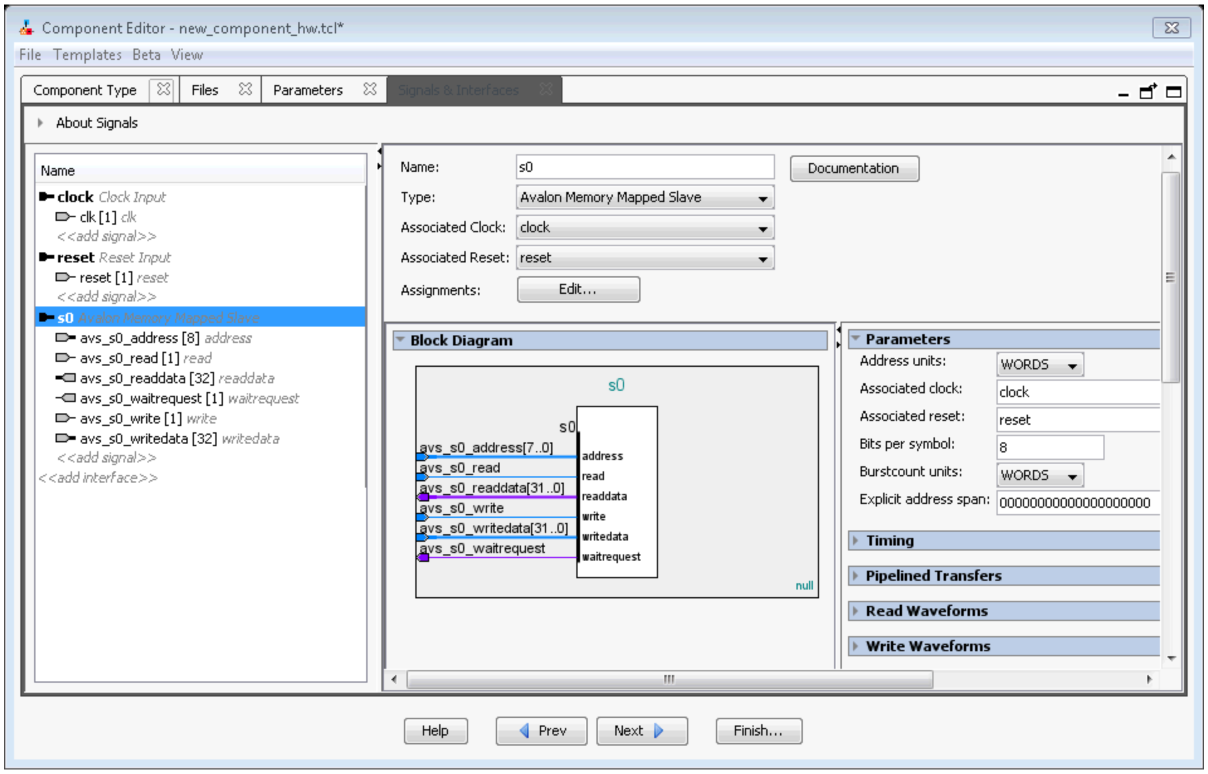
Specify Interfaces and Signals in the Qsys Component Editor

The **Signals & Interfaces** tab allows you to add signals and interfaces to your component.

You can configure the type and properties of signals and interfaces. Some interfaces display waveforms that illustrate the timing of the interface. If you update the timing parameters, the waveforms automatically update.

1. In Qsys, click **new_component** in the IP Catalog.
2. In the Qsys Component Editor, click the **Signals & Interfaces** tab.
3. To create an interface using a template, click any template in the Templates drop-down menu, and then perform the following steps:
   a. Click each bold interface to edit its properties in the right pane.
   b. For each interface, click the its associated signals to edit their properties in the right pane. The signals for an interface appear as indented elements below the interface.

Signals & Interfaces tab - Templates



4. To create a custom interface and its associated signals, in the **Signals & Interface**s tab, perform the following steps:
   a. Click <*add interface*>, and then click inside the left pane.
   b. To create an associated signal for the new interface, click <*add signal*> under the new interface, and then click inside the left pane.
   c. Click each bold interface to edit its properties in the right pane.
   d. For each interface, click its associated signals to edit their properties in the right pane.
5. To move signals between interfaces, select the signal, and then drag it to another interface.
6. To rename an interface or signal, select the element, and then press **F2**.
7. To remove an interface or signal, right-click the element, and then click **Remove**.
   Alternatively, to remove an interface or signal, you can select the element, and then press **Delete**. When you remove an interface, Qsys also removes all of it associated signals.

Specify Parameters in the Qsys Component Editor

Components can include parameterized HDL, which allow users of the component flexibility in meeting their system requirements. For example, a component may have a configurable memory size or data width, where one HDL implementation can be used in different systems, each with unique parameters values.

The **Parameters** tab allows you specify the parameters that are used to configure instances of the component in a Qsys system. You can specify various properties for each parameter that describe how to display and use the parameter. You can also specify a range of allowed values that are checked during the validation phase. The **Parameters** table displays the HDL parameters that are declared in the top-level HDL module. If you have not yet created the top-level HDL file, the parameters that you create on the **Parameters** tab are included in the top-level synthesis file template created from the **Files** tab.

When the component includes HDL files, the parameters match those defined in the top-level module, and you cannot be add or remove them on the **Parameters** tab. To add or remove the parameters, edit your HDL source, and then re-analyze the file.

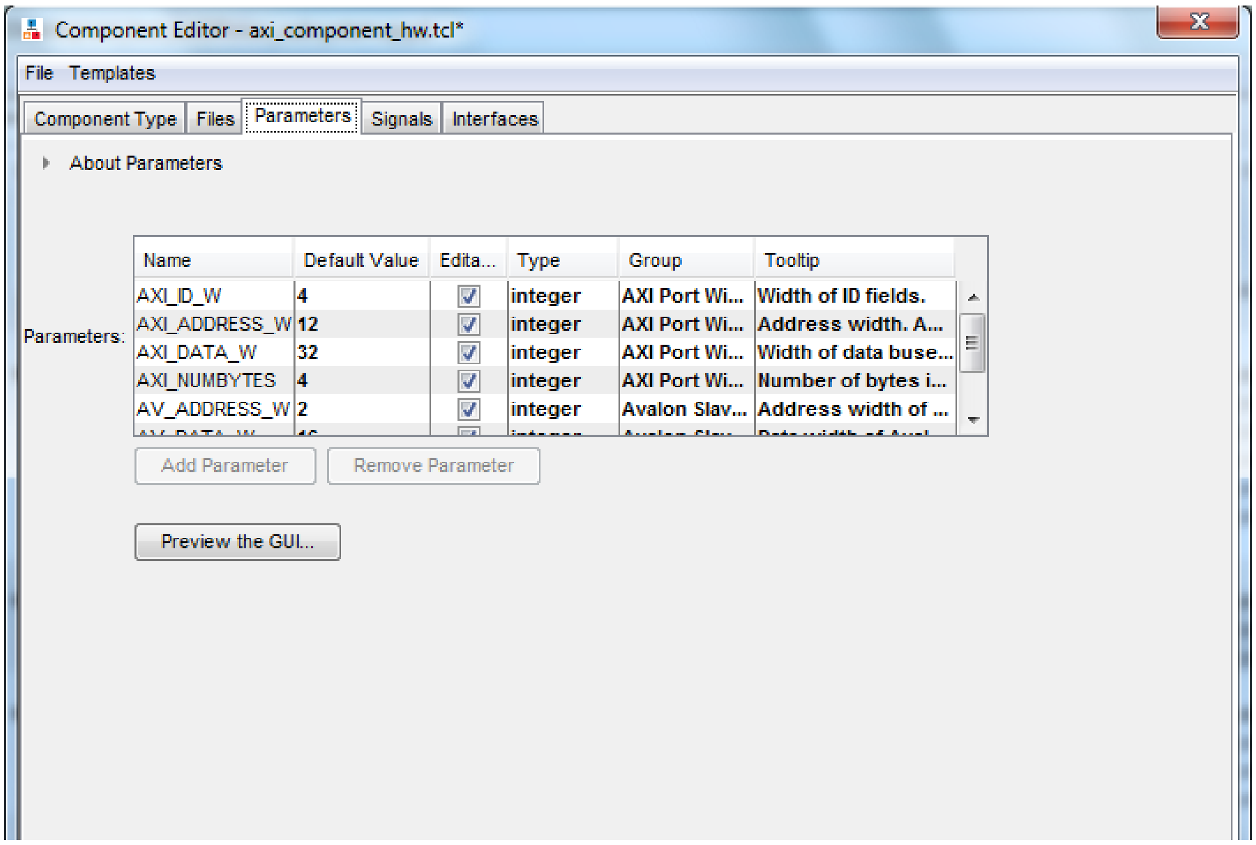
If you used the Component Editor to create a top-level template HDL file for synthesis, you can remove the newly-created file from the **Synthesis Files** list on the **Files** tab, make your parameter changes, and then re-analyze the top-level synthesis file.

You can use the **Parameters** table to specify the following information about each parameter:

- **Name**—Specifies the name of the parameter.
- **Default Value**—Sets the default value used in new instances of the component.
- **Editable**—Specifies whether or not the user can edit the parameter value.
- **Type**—Defines the parameter type as string, integer, boolean, std_logic, logic vector, natural, or positive.
- **Group**—Allows you to group parameters in parameter editor.
- **Tooltip**—Allows you to add a description of the parameter that appears when the user of the component points to the parameter in the parameter editor.

Figure 6-4: Parameters Tab in the Qsys Components Editor

On the **Parameters** tab, you can click **Preview the GUI** at any time to see how the declared parameters appear in the parameter editor. Parameters with their default values appear with checks in the **Editable** column, indicating that users of this component are allowed to modify the parameter value. Editable parameters cannot contain computed expressions. You can group parameters under a common heading or section in the parameter editor with the **Group** column, and a tooltip helps users of the component understand the function of the parameter. Various parameter properties allow you to customize the component's parameter editor, such as using radio buttons for parameter selections, or displaying an image.



Name	Default Value	Edita...	Type	Group	Tooltip
AXI_ID_W	4	✓	integer	AXI Port Wi...	Width of ID fields.
AXI_ADDRESS_W	12	✓	integer	AXI Port Wi...	Address width. A...
AXI_DATA_W	32	✓	integer	AXI Port Wi...	Width of data buse...
AXI_NUMBYTES	4	✓	integer	AXI Port Wi...	Number of bytes i...
AV_ADDRESS_W	2	✓	integer	Avalon Slav...	Address width of ...

Example 6-6: _hw.tcl Created from Entries in the Parameters Tab

In this example, the first `add_parameter` command includes commonly-specified properties. The `set_parameter_property` command specifies each property individually. The **Tooltip** column on the **Parameters** tab maps to the `DESCRIPTION` property, and there is an additional unused `UNITS` property created in the code. The `HDL_PARAMETER` property specifies that the value of the parameter is specified in the HDL instance wrapper when creating instances of the component. The **Group** column in the **Parameters** tab maps to the display items section with the `add_display_item` commands.

Note: If a parameter <*n*> defines the width of a signal, the signal width must follow the format: <*n*-1>:0.

```
#
# parameters
#
add_parameter AXI_ID_W INTEGER 4 "Width of ID fields"
set_parameter_property AXI_ID_W DEFAULT_VALUE 4
set_parameter_property AXI_ID_W DISPLAY_NAME AXI_ID_W
set_parameter_property AXI_ID_W TYPE INTEGER
set_parameter_property AXI_ID_W UNITS None
set_parameter_property AXI_ID_W DESCRIPTION "Width of ID fields"
set_parameter_property AXI_ID_W HDL_PARAMETER true
add_parameter AXI_ADDRESS_W INTEGER 12
set_parameter_property AXI_ADDRESS_W DEFAULT_VALUE 12

add_parameter AXI_DATA_W INTEGER 32
...
#
# display items
#
add_display_item "AXI Port Widths" AXI_ID_W PARAMETER ""
```

Note: If an AXI slave's ID bit width is smaller than required for your system, the AXI slave response may not reach all AXI masters. The formula of an AXI slave ID bit width is calculated as follows:

`maximum_master_id_width_in_the_interconnect + log2` (number_of_masters_in_the_same_interconnect)

For example, if an AXI slave connects to three AXI masters and the maximum AXI master ID length of the three masters is 5 bits, then the AXI slave ID is 7 bits, and is calculated as follows:

5 bits + 2 bits ($\log_2$(3 masters)) = 7

Table 6-3: AXI Master and Slave Parameters

Qsys refers to AXI interface parameters to build AXI interconnect. If these parameter settings are incompatible with the component's HDL behavior, Qsys interconnect and transactions may not work correctly. To prevent unexpected interconnect behavior, you must set the AXI component parameters.

AXI Master Parameters	AXI Slave Parameters
`readIssuingCapability`	`readAcceptanceCapability`
`writeIssuingCapability`	`writeAcceptanceCapability`
`combinedIssuingCapability`	`combinedAcceptanceCapability`
	`readDataReorderingDepth`

Related Information

- **Component Interface Tcl Reference** on page 9-1

Valid Ranges for Parameters in the _hw.tcl File

In the **_hw.tcl** file, you can specify valid ranges for parameters.

Qsys validation checks each parameter value against the `ALLOWED_RANGES` property. If the values specified are outside of the allowed ranges, Qsys displays an error message. Specifying choices for the allowed values enables users of the component to choose the parameter value from a drop-down list or radio button in the parameter editor GUI instead of entering a value.

The `ALLOWED_RANGES` property is a list of valid ranges, where each range is a single value, or a range of values defined by a start and end value.

Table 6-4: ALLOWED_RANGES Property

ALLOWED_RANGES Property	Values
`{a b c}`	a, b, or c
`{"No Control" "Single Control" "Dual Controls"}`	Unique string values. Quotation marks are required if the strings include spaces .
`{1 2 4 8 16}`	1, 2, 4, 8, or 16
`{1:3}`	1 through 3, inclusive.
`{1 2 3 7:10}`	1, 2, 3, or 7 through 10 inclusive.

Related Information

Declare Parameters with Custom _hw.tcl Commands on page 6-23

Types of Qsys Parameters

Qsys uses the following parameter types: user parameters, system information parameters, and derived parameters.

Qsys User Parameters on page 6-22

Qsys System Information Parameters on page 6-22

Qsys Derived Parameters on page 6-23

Related Information

Declare Parameters with Custom _hw.tcl Commands on page 6-23

Qsys User Parameters

User parameters are parameters that users of a component can control, and appear in the parameter editor for instances of the component. User parameters map directly to parameters in the component HDL. For user parameter code examples, such as `AXI_DATA_W` and `ENABLE_STREAM_OUTPUT`, refer to *Declaring Parameters with Custom hw.tcl Commands.*

Qsys System Information Parameters

A `SYSTEM_INFO` parameter is a parameter whose value is set automatically by the Qsys system. When you define a `SYSTEM_INFO` parameter, you provide an `information type,` and additional arguments.

For example, you can configure a parameter to store the clock frequency driving a clock input for your component. To do this, define the parameter as `SYSTEM_INFO` of type `CLOCK_RATE`:

```
set_parameter_property <param> SYSTEM_INFO CLOCK_RATE
```

You then set the name of the clock interface as the `SYSTEM_INFO` argument:

```
set_parameter_property <param> SYSTEM_INFO_ARG <clkname>
```

Qsys Derived Parameters

Derived parameter values are calculated from other parameters during the Elaboration phase, and are specified in the **hw.tcl** file with the `DERIVED` property. Derived parameter values are calculated from other parameters during the Elaboration phase, and are specified in the **hw.tcl** file with the `DERIVED` property. For example, you can derive a clock period parameter from a data rate parameter. Derived parameters are sometimes used to perform operations that are difficult to perform in HDL, such as using logarithmic functions to determine the number of address bits that a component requires.

Related Information

Declare Parameters with Custom _hw.tcl Commands on page 6-23

Parameterized Parameter Widths

Qsys allows a `std_logic_vector` parameter to have a width that is defined by another parameter, similar to derived parameters. The width can be a constant or the name of another parameter.

Declare Parameters with Custom _hw.tcl Commands

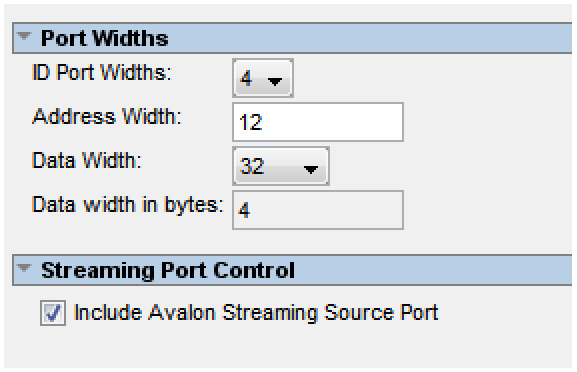
The example below illustrates a custom **_hw.tcl** file, with more advanced parameter commands than those generated when you specify parameters in the Component Editor. Commands include the `ALLOWED_RANGES` property to provide a range of values for the `AXI_ADDRESS_W` (**Address Width**) parameter, and a list of parameter values for the `AXI_DATA_W` (**Data Width**) parameter. This example also shows the parameter `AXI_NUMBYTES` (**Data width in bytes**) parameter; that uses the `DERIVED` property. In addition, these commands illustrate the use of the `GROUP` property, which groups some parameters under a heading in the parameter editor GUI. You use the `ENABLE_STREAM_OUTPUT_GROUP` (**Include Avalon streaming source port**) parameter to enable or disable the optional Avalon-ST interface in this design, and is displayed as a check box in the parameter editor GUI because the parameter is of type BOOLEAN. Refer to figure below to see the parameter editor GUI resulting from these **hw.tcl** commands.

Example 6-7: Parameter Declaration

In this example, the `AXI_NUMBYTES` parameter is derived during the Elaboration phase based on another parameter, instead of being assigned to a specific value. `AXI_NUMBYTES` describes the number of bytes in a word of data. Qsys calculates the `AXI_NUMBYTES` parameter from the `DATA_WIDTH` parameter by dividing by 8. The **_hw.tcl** code defines the `AXI_NUMBYTES` parameter as a derived parameter, since its value is calculated in an elaboration callback procedure. The `AXI_NUMBYTES` parameter value is not editable, because its value is based on another parameter value.

```
add_parameter AXI_ADDRESS_W INTEGER 12

set_parameter_property AXI_ADDRESS_W DISPLAY_NAME \
"AXI Slave Address Width"

set_parameter_property AXI_ADDRESS_W DESCRIPTION \
"Address width."

set_parameter_property AXI_ADDRESS_W UNITS bits
set_parameter_property AXI_ADDRESS_W ALLOWED_RANGES 4:16
set_parameter_property AXI_ADDRESS_W HDL_PARAMETER true

set_parameter_property AXI_ADDRESS_W GROUP \
"AXI Port Widths"

add_parameter AXI_DATA_W INTEGER 32
set_parameter_property AXI_DATA_W DISPLAY_NAME "Data Width"

set_parameter_property AXI_DATA_W DESCRIPTION \
"Width of data buses."

set_parameter_property AXI_DATA_W UNITS bits

set_parameter_property AXI_DATA_W ALLOWED_RANGES \
{8 16 32 64 128 256 512 1024}

set_parameter_property AXI_DATA_W HDL_PARAMETER true
set_parameter_property AXI_DATA_W GROUP "AXI Port Widths"

add_parameter AXI_NUMBYTES INTEGER 4
set_parameter_property AXI_NUMBYTES DERIVED true

set_parameter_property AXI_NUMBYTES DISPLAY_NAME \
"Data Width in bytes; Data Width/8"

set_parameter_property AXI_NUMBYTES DESCRIPTION \
"Number of bytes in one word"

set_parameter_property AXI_NUMBYTES UNITS bytes
set_parameter_property AXI_NUMBYTES HDL_PARAMETER true
set_parameter_property AXI_NUMBYTES GROUP "AXI Port Widths"

add_parameter ENABLE_STREAM_OUTPUT BOOLEAN true

set_parameter_property ENABLE_STREAM_OUTPUT DISPLAY_NAME \
"Include Avalon Streaming Source Port"

set_parameter_property ENABLE_STREAM_OUTPUT DESCRIPTION \
"Include optional Avalon-ST source (default),\
or hide the interface"

set_parameter_property ENABLE_STREAM_OUTPUT GROUP \
"Streaming Port Control"

...
```

Figure 6-5: Resulting Parameter Editor GUI from Parameter Declarations



Related Information

Control Interfaces Dynamically with an Elaboration Callback on page 6-26

Component Interface Tcl Reference on page 9-1

Validate Parameter Values with a Validation Callback

You can use a validation callback procedure to validate parameter values with more complex validation operations than the `ALLOWED_RANGES` property allows. You define a validation callback by setting the `VALIDATION_CALLBACK` module property to the name of the Tcl callback procedure that runs during the validation phase. In the validation callback procedure, the current parameter values is queried, and warnings or errors are reported about the component's configuration.

Example 6-8: Demo AXI Memory Example

If the optional Avalon streaming interface is enabled, then the control registers must be wide enough to hold an AXI RAM address, so the designer can add an error message to ensure that the user enters allowable parameter values.

```
set_module_property VALIDATION_CALLBACK validate
proc validate {} {
if {
  [get_parameter_value ENABLE_STREAM_OUTPUT ] &&
  ([get_parameter_value AXI_ADDRESS_W] >
  [get_parameter_value AV_DATA_W])
}
send_message error "If the optional Avalon streaming port\
is enabled, the AXI Data Width must be equal to or greater\
than the Avalon control port Address Width"
}
}
```

Related Information

Component Interface Tcl Reference on page 9-1

Demo AXI Memory Example

Control Interfaces Dynamically with an Elaboration Callback

You can allow user parameters to dynamically control your component's behavior with a an elaboration callback procedure during the elaboration phase. Using an elaboration callback allows you to change interface properties, remove interfaces, or add new interfaces as a function of a parameter value. You define an elaboration callback by setting the module property `ELABORATION_CALLBACK` to the name of the Tcl callback procedure that runs during the elaboration phase. In the callback procedure, you can query the parameter values of the component instance, and then change the interfaces accordingly.

Example 6-9: Avalon-ST Source Interface Optionally Included in a Component Specified with an Elaboration Callback

```
set_module_property ELABORATION_CALLBACK elaborate

proc elaborate {} {

   # Optionally disable the Avalon- ST data output

    if{[ get_parameter_value ENABLE_STREAM_OUTPUT] == "false" }{
       set_port_property aso_data    termination true
       set_port_property aso_valid   termination true
       set_port_property aso_ready   termination true
       set_port_property aso_ready   termination_value 0
    }
  # Calculate the Data Bus Width in bytes

    set bytewidth_var [expr [get_parameter_value AXI_DATA_W]/8]
    set_parameter_value AXI_NUMBYTES $bytewidth_var
}
```

Related Information

- **Creating Custom _hw.tcl Interface Settings and Properties**
- **Validate Parameter Values with a Validation Callback** on page 6-25
- **Component Interface Tcl Reference** on page 9-1

Control File Generation Dynamically with Parameters and a Fileset Callback

You can use a fileset callback to control which files are created in the output directories during the generation phase based on parameter values, instead of providing a fixed list of files. In a callback procedure, you can query the values of the parameters and use them to generate the appropriate files. To define a fileset callback, you specify a callback procedure name as an argument in the `add_fileset` command. You can use the same fileset callback procedure for all of the filesets, or create separate procedures for synthesis and simulation, or Verilog and VHDL.

Example 6-10: Fileset Callback Using Parameters to Control Filesets in Two Different Ways

The `RAM_VERSION` parameter chooses between two different source files to control the implementation of a RAM block. For the top-level source file, a custom Tcl routine generates HDL that optionally includes control and status registers, depending on the value of the `CSR_ENABLED` parameter.

During the generation phase, Qsys creates a a top-level Qsys system HDL wrapper module to instantiate the component top-level module, and applies the component's parameters, for any parameter whose parameter property `HDL_PARAMETER` is set to true.

```
#Create synthesis fileset with fileset_callback and set top level

add_fileset my_synthesis_fileset QUARTUS_SYNTH fileset_callback

set_fileset_property my_synthesis_fileset TOP_LEVEL \
demo_axi_memory

# Create Verilog simulation fileset with same fileset_callback
# and set top level

add_fileset my_verilog_sim_fileset SIM_VERILOG fileset_callback

set_fileset_property my_verilog_sim_fileset TOP_LEVEL \
demo_axi_memory

# Add extra file needed for simulation only

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv

# Create VHDL simulation fileset (with Verilog files
# for mixed-language VHDL simulation)

add_fileset my_vhdl_sim_fileset SIM_VHDL fileset_callback
set_fileset_property my_vhdl_sim_fileset TOP_LEVEL demo_axi_memory

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH
verification_lib/verbosity_pkg.sv

# Define parameters required for fileset_callback

add_parameter RAM_VERSION INTEGER 1
set_parameter_property RAM_VERSION ALLOWED_RANGES {1 2}
set_parameter_property RAM_VERSION HDL_PARAMETER false
add_parameter CSR_ENABLED BOOLEAN enable
set_parameter_property CSR_ENABLED HDL_PARAMETER false

# Create Tcl callback procedure to add appropriate files to
# filesets based on parameters

proc fileset_callback { entityName } {
  send_message INFO "Generating top-level entity $entityName"
  set ram [get_parameter_value RAM_VERSION]
  set csr_enabled [get_parameter_value CSR_ENABLED]

  send_message INFO "Generating memory
  implementation based on RAM_VERSION $ram    "

      if {$ram == 1} {
          add_fileset_file single_clk_ram1.v VERILOG PATH \
    single_clk_ram1.v
      } else     {
          add_fileset_file single_clk_ram2.v VERILOG PATH \
    single_clk_ram2.v
    }

send_message INFO "Generating top-level file for \
CSR_ENABLED $csr_enabled"
generate_my_custom_hdl $csr_enabled demo_axi_memory_gen.sv

add_fileset_file demo_axi_memory_gen.sv VERILOG PATH \
demo_axi_memory_gen.sv
}
```

Related Information

Specify Synthesis and Simulation Files in the Qsys Component Editor on page 6-12

Component Interface Tcl Reference on page 9-1

Create a Composed Component or Subsystem

A composed component is a subsystem containing instances of other components. Unlike an HDL-based component, a composed component's HDL is created by generating HDL for the components in the subsystem, in addition to the Qsys interconnect to connect the subsystem instances.

You can add child instances in a composition callback of the **_hw.tcl** file.

With a composition callback, you can also instantiate and parameterize sub-components as a function of the composed component’s parameter values. You define a composition callback by setting the `COMPOSITION_CALLBACK` module property to the name of the composition callback procedures.

A composition callback replaces the validation and elaboration phases. HDL for the subsystem is generated by generating all of the sub-components and the top-level that combines them.

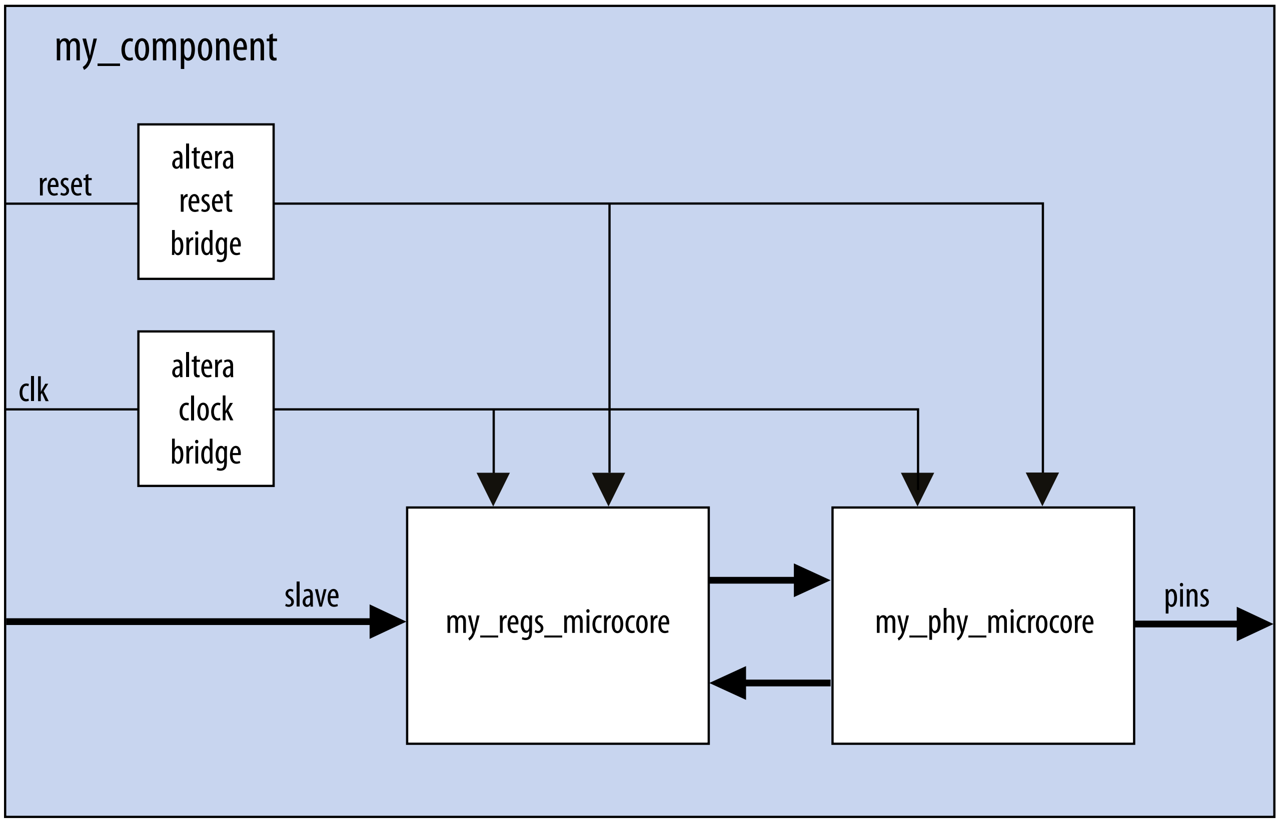
To connect instances of your component, you must define the component's interfaces. Unlike an HDL-based component, a composed component does not directly specify the signals that are exported. Instead, interfaces of submodules are chosen as the external interface, and each internal interface's ports are connected through the exported interface.

Exporting an interface means that you are making the interface visible from the outside of your component, instead of connecting it internally. You can set the `EXPORT_OF` property of the externally visible interface from the main program or the composition callback, to indicate that it is an exported view of the submodule's interface.

Exporting an interface is different than defining an interface. An exported interface is an exact copy of the subcomponent’s interface, and you are not allowed to change properties on the exported interface. For example, if the internal interface is a 32-bit or 64-bit master without bursting, then the exported interface is the same. An interface on a subcomponent cannot be exported and also connected within the subsystem.

When you create an exported interface, the properties of the exported interface are copied from the subcomponent’s interface without modification. Ports are copied from the subcomponent’s interface with only one modification; the names of the exported ports on the composed component are chosen to ensure that they are unique.

Figure 6-6: Top-Level of a Composed Component



Example 6-11: Composed _hw.tcl File that Instantiates Two Sub-Components

Qsys connects the components, and also connects the clocks and resets. Note that clock and reset bridge components are required to allow both sub-components to see common clock and reset inputs.

```
package require -exact qsys 14.0
set_module_property name my_component
set_module_property COMPOSITION_CALLBACK composed_component

proc composed_component {} {
  add_instance clk altera_clock_bridge
  add_instance reset altera_reset_bridge
  add_instance regs my_regs_microcore
  add_instance phy my_phy_microcore

  add_interface clk clock end
  add_interface reset reset end
  add_interface slave avalon slave
  add_interface pins conduit end

  set_interface_property clk EXPORT_OF clk.in_clk
  set_instance_property_value reset synchronous_edges deassert
  set_interface_property reset EXPORT_OF reset.in_reset
  set_interface_property slave EXPORT_OF regs.slave
  set_interface_property pins  EXPORT_OF phy.pins
```

```
   add_connection clk.out_clk reset.clk
   add_connection clk.out_clk regs.clk
   add_connection clk.out_clk phy.clk
   add_connection reset.out_reset regs.reset
   add_connection reset.out_reset phy.clk_reset
   add_connection regs.output phy.input
   add_connection phy.output regs.input
 }
```

Related Information

- **Component Interface Tcl Reference** on page 9-1

Create an IP Component with Qsys a System View Different from the Generated Synthesis Output Files

There are cases where it may be beneficial to have the structural Qsys system view of a component differ from the generated synthesis output files. The structural composition callback allows you to define a structural hierarchy for a component separately from the generated output files.

One application of this feature is for IP designers who want to send out a placed-and-routed component that represents a Qsys system in order to ensure timing closure for the end-user. In this case, the designer creates a design partition for the Qsys system, and then exports a post-fit Quartus II Exported Partition File (**.qxp**) when satisfied with the placement and routing results.

The designer specifies a **.qxp** file as the generated synthesis output file for the new component. The designer can specify whether to use a simulation output fileset for the custom simulation model file, or to use simulation output files generated from the original Qsys system.

When the end-user adds this component to their Qsys system, the designer wants the end-user to see a structural representation of the component, including lower-level components and the address map of the original Qsys system. This structural view is a logical representation of the component that is used during the elaboration and validation phases in Qsys.

Example 6-12: Structural Composition Callback and .qxp File as the Generated Output

To specify a structural representation of the component for Qsys, connect components or generate a hardware Tcl description of the Qsys system, and then insert the Tcl commands into a structural composition callback. To invoke the structural composition callback use the command:

```
set_module_property STRUCTURAL_COMPOSITION_CALLBACK structural_hierarchy
```

```
 package require -exact qsys 14.0
 set_module_property name example_structural_composition

 set_module_property STRUCTURAL_COMPOSITION_CALLBACK \
 structural_hierarchy

 add_fileset synthesis_fileset QUARTUS_SYNTH \
 synth_callback_procedure

 add_fileset simulation_fileset SIM_VERILOG \
 sim_callback_procedure

 set_fileset_property synthesis_fileset TOP_LEVEL \
 my_custom_component
```

```
set_fileset_property simulation_fileset TOP_LEVEL \
my_custom_component

proc structural_hierarchy {} {

# called during elaboration and validation phase
# exported ports should be same in structural_hierarchy
# and generated QXP

# These commands could come from the exported hardware Tcl

    add_interface clk clock sink
    add_interface reset reset sink

    add_instance clk_0 clock_source
    set_interface_property clk EXPORT_OF clk_0.clk_in
    set_interface_property reset EXPORT_OF clk_0.clk_in_reset

    add_instance pll_0 altera_pll
    # connections and connection parameters
    add_connection clk_0.clk pll_0.refclk clock
    add_connection clk_0.clk_reset pll_0.reset reset
}

proc synth_callback_procedure { entity_name } {

# the QXP should have the same name for ports
# as exportedin structural_hierarchy

 add_fileset_file my_custom_component.qxp QXP PATH \
 "my_custom_component.qxp"
}

proc sim_callback_procedure { entity_name } {

# the simulation files should have the same name for ports as
# exported in structural_hierarchy

add_fileset_file my_custom_component.v VERILOG PATH \
"my_custom_component.v"
 ….
 ….
}
```

Related Information

Create a Composed Component or Subsystem on page 6-28

Add Component Instances to a Static or Generated Component

You can create nested components by adding component instances to an existing component. Both static and generated components can create instances of other components. You can add child instances of a component in a **_hw.tcl** using elaboration callback.

With an elaboration callback, you can also instantiate and parameterize sub-components with the `add_hdl_instance` command as a function of the parent component's parameter values.

When you instantiate multiple nested components, you must create a unique variation name for each component with the `add_hdl_instance` command. Prefixing a variation name with the parent component name prevents conflicts in a system. The variation name can be the same across multiple parent components if the generated parameterization of the nested component is exactly the same.

Note: If you do not adhere to the above naming variation guidelines, Qsys validation-time errors occur, which are often difficult to debug.

Related Information

- **Static Components** on page 6-32
- **Generated Components** on page 6-33

Static Components

Static components always generate the same output, regardless of their parameterization. Components that instantiate static components must have only static children.

A design file that is static between all parameterizations of a component can only instantiate other static design files. Since static IPs always render the same HDL regardless of parameterization, Qsys generates static IPs only once across multiple instantiations, meaning they have the same top-level name set.

Example 6-13: Typical Usage of the add_hdl_instance Command for Static Components

```
package require -exact qsys 14.0

set_module_property name add_hdl_instance_example
add_fileset synth_fileset QUARTUS_SYNTH synth_callback
set_fileset_property synth_fileset TOP_LEVEL basic_static
set_module_property elaboration_callback elab

proc elab {} {
  # Actual API to instantiate an IP Core
  add_hdl_instance emif_instance_name altera_mem_if_ddr3_emif

  # Make sure the parameters are set appropriately
  set_instance_parameter_value emif_instance_name SPEED_GRADE {7}
  ...
  }
proc synth_callback { output_name } {
  add_fileset_file "basic_static.v" VERILOG PATH basic_static.v
}
```

Example 6-14: Top-Level HDL Instance and Wrapper File Created by Qsys

In this example, Qsys generates a wrapper file for the instance name specified in the **_hw.tcl** file.

```
//Top Level Component HDL
module basic_static (input_wire, output_wire, inout_wire);
input [31:0] input_wire;
output [31:0] output_wire;
inout [31:0] inout_wire;

// Instantiation of the instance added via add_hdl_instance
// command. This is an example of how the instance added via
// the add_hdl_instance command can be used
// in the top-level file of the component.

emif_instance_name fixed_name_instantiation_in_top_level(
.pll_ref_clk (input_wire), // pll_ref_clk.clk
.global_reset_n (input_wire), // global_reset.reset_n
.soft_reset_n (input_wire), // soft_reset.reset_n
...
... );
endmodule

//Wrapper for added HDL instance
// emif_instance_name.v
// Generated using ACDS version 14.0

`timescale 1 ps / 1 ps
module emif_instance_name (
input wire pll_ref_clk, // pll_ref_clk.clk
input wire global_reset_n, // global_reset.reset_n
input wire soft_reset_n, // soft_reset.reset_n
output wire afi_clk, // afi_clk.clk
...
...);
example_addhdlinstance_system
_add_hdl_instance_example_0_emif_instance
_name_emif_instance_name emif_instance_name (

.pll_ref_clk (pll_ref_clk), // pll_ref_clk.clk
.global_reset_n (global_reset_n), // global_reset.reset_n
.soft_reset_n (soft_reset_n), // soft_reset.reset_n
...
...);
endmodule
```

Generated Components

A generated component's fileset callback allows an instance of the component to create unique HDL design files based on the instance's parameter values. For example, you can write a fileset callback to include a control and status interface based on the value of a parameter. The callback overcomes a limitation of HDL languages, which do not allow run-time parameters.

Generated components change their generation output (HDL) based on their parameterization. If a component is generated, then any component that may instantiate it with multiple parameter sets must also be considered generated, since its HDL changes with its parameterization. This case has an effect that propagates up to the top-level of a design.

Since generated components are generated for each unique parameterized instantiation, when implementing the `add_hdl_instance` command, you cannot use the same fixed name (specified using `instance_name`) for the different variants of the child HDL instances. To facilitate unique naming for the wrapper of each unique parameterized instantiation of child HDL instances, you must use the following command so that Qsys generates a unique name for each wrapper. You can then access this unique wrapper name with a fileset callback so that the instances are instantiated inside the component's top-level HDL.

- To declare auto-generated fixed names for wrappers, use the command:

  ```
  set_instance_property instance_name HDLINSTANCE_USE_GENERATED_NAME \
  true
  ```

  Note: You can only use this command with a generated component in the global context, or in an elaboration callback.

- To obtain auto-generated fixed name with a fileset callback, use the command:

  ```
  get_instance_property instance_name HDLINSTANCE_GET_GENERATED_NAME
  ```

  Note: You can only use this command with a fileset callback. This command returns the value of the auto-generated fixed name, which you can then use to instantiate inside the top-level HDL.

Example 6-15: Typical Usage of the add_hdl_instance Command for Generated Components

Qsys generates a wrapper file for the instance name specified in the **_hw.tcl** file.

```
package require -exact qsys 14.0
set_module_property name generated_toplevel_component
set_module_property ELABORATION_CALLBACK elaborate
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH generate
add_fileset SIM_VERILOG SIM_VERILOG generate
add_fileset SIM_VHDL SIM_VHDL generate

proc elaborate {} {

 # Actual API to instantiate an IP Core
 add_hdl_instance emif_instance_name altera_mem_if_ddr3_emif

 # Make sure the parameters are set appropriately
 set_instance_parameter_value emif_instance_name SPEED_GRADE {7}
 ...
 # instruct Qsys to use auto generated fixed name
 set_instance_property emif_instance_name \
 HDLINSTANCE_USE_GENERATED_NAME 1
}

proc generate { entity_name } {

 # get the autogenerated name for emif_instance_name added
 # via add_hdl_instance

 set autogeneratedfixedname [get_instance_property \
 emif_instance_name HDLINSTANCE_GET_GENERATED_NAME]

 set fileID [open "generated_toplevel_component.v" r]
 set temp ""

 # read the contents of the file

 while {[eof $fileID] != 1} {
 gets $fileID lineInfo

 # replace the top level entity name with the name provided
 # during generation

 regsub -all "substitute_entity_name_here" $lineInfo \
 "${entity_name}" lineInfo

 # replace the autogenerated name for emif_instance_name added
 # via add_hdl_instance
```

```
     regsub -all "substitute_autogenerated_emifinstancename_here" \
     $lineInfo"${autogeneratedfixedname}" lineInfo \
     append temp "${lineInfo}\n"
    }

    # adding a top level component file

    add_fileset_file ${entity_name}.v VERILOG TEXT $temp
    }
```

Example 6-16: Top-Level HDL Instance and Wrapper File Created By Qsys

```
// Top Level Component HDL

module substitute_entity_name_here (input_wire, output_wire,
inout_wire);

input [31:0] input_wire;
output [31:0] output_wire;
inout [31:0] inout_wire;

// Instantiation of the instance added via add_hdl_instance
// command. This is an example of how the instance added
// via add_hdl_instance command can be used
// in the top-level file of the component.

substitute_autogenerated_emifinstancename_here
fixed_name_instantiation_in_top_level (
.pll_ref_clk (input_wire), // pll_ref_clk.clk
.global_reset_n (input_wire), // global_reset.reset_n
.soft_reset_n (input_wire), // soft_reset.reset_n
...
... );
endmodule

// Wrapper for added HDL instance
// generated_toplevel_component_0_emif_instance_name.v is the
// auto generated //emif_instance_name
// Generated using ACDS version 13.

`timescale 1 ps / 1 ps
module generated_toplevel_component_0_emif_instance_name (
input wire pll_ref_clk, // pll_ref_clk.clk
input wire global_reset_n, // global_reset.reset_n
input wire soft_reset_n, // soft_reset.reset_n
output wire afi_clk, // afi_clk.clk
...
...);
example_addhdlinstance_system_add_hdl_instance_example_0_emif
_instance_name_emif_instance_name emif_instance_name (

.pll_ref_clk (pll_ref_clk), // pll_ref_clk.clk
.global_reset_n (global_reset_n), // global_reset.reset_n
.soft_reset_n (soft_reset_n), // soft_reset.reset_n
...
...);
endmodule
```

Related Information

Control File Generation Dynamically with Parameters and a Fileset Callback on page 6-26

Design Guidelines for Adding Component Instances

In order to promote standard and predictable results when generating static and generated components, Altera recommends the following best-practices:

- For two different parameterizations of a component, a component must never generate a file of the same name with different instantiations. The contents of a file of the same name must be identical for every parameterization of the component.
- If a component generates a nested component, it must never instantiate two different parameterizations of the nested component using the same instance name. If the parent component's parameterization affects the parameters of the nested component, the parent component must use a unique instance name for each unique parameterization of the nested component.
- Static components that generate differently based on parameterization have the potential to cause problems in the following cases:
  - Different file names with the same entity names, results in same entity conflicts at compilation-time
  - Different contents with the same file name results in overwriting other instances of the component, and in either file, compile-time conflicts or unexpected behavior.
- Generated components that generate files not based on the output name and that have different content results in either compile-time conflicts, or unexpected behavior.

Document Revision History

The table below indicates edits made to the *Creating Qsys Components* content since its creation.

Table 6-5: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• Updated screen shots **Files** tab, Qsys Component Editor. • Added topic: *Specify Interfaces and Signals in the Qsys Component Editor*. • Added topic: *Create an HDL File in the Qsys Component Editor*. • Added topic: *Create an HDL File Using a Template in the Qsys Component Editor*.
November 2013	13.1.0	• `add_hdl_instance` • Added *Creating a Component With Differing Structural Qsys View and Generated Output Files*.

Date	Version	Changes
May 2013	13.0.0	• Consolidated content from other Qsys chapters. • Added *Upgrading IP Components to the Latest Version*. • Updated for AMBA APB support.
November 2012	12.1.0	• Added AMBA AXI4 support. • Added the **demo_axi_memory** example with screen shots and example **_hw.tcl** code.
June 2012	12.0.0	• Added new tab structure for the Component Editor. • Added AMBA AXI3 support.
November 2011	11.1.0	Template update.
May 2011	11.0.0	• Removed beta status. • Added Avalon Tri-state Conduit (Avalon-TC) interface type. • Added many interface templates for Nios custom instructions and Avalon-TC interfaces.
December 2010	10.1.0	Initial release.

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

Related Information

Quartus II Handbook Archive

Qsys Interconnect 7

2015.05.04

QII5V1 Subscribe Send Feedback

Qsys interconnect is a high-bandwidth structure that allows you to connect IP components to other IP components with various interfaces.

Qsys supports Avalon, AMBA AXI3 (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB3 (version 1.0) interface specifications.

Note: The video, *AMBA AXI and Altera Avalon Interoperation Using Qsys*, describes seamless integration of IP components using the AMBA AXI interface, and the Altera Avalon interface.

Related Information

- **Avalon Interface Specifications**
- **AMBA Specifications**
- **Creating a System with Qsys** on page 5-1
- **Creating Qsys Components** on page 6-1
- **Qsys System Design Components** on page 10-1
- **AMBA AXI and Altera Avalon Interoperation Using Qsys**

Memory-Mapped Interfaces

Qsys supports the implementation of memory-mapped interfaces for Avalon, AXI, and APB protocols.

Qsys interconnect transmits memory-mapped transactions between masters and slaves in packets. The command network transports read and write packets from master interfaces to slave interfaces. The response network transports response packets from slave interfaces to master interfaces.

For each component interface, Qsys interconnect manages memory-mapped transfers and interacts with signals on the connected interface. Master and slave interfaces can implement different signals based on interface parameterizations, and Qsys interconnect provides any necessary adaptation between them. In the path between master and slaves, Qsys interconnect may introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the interfaces.

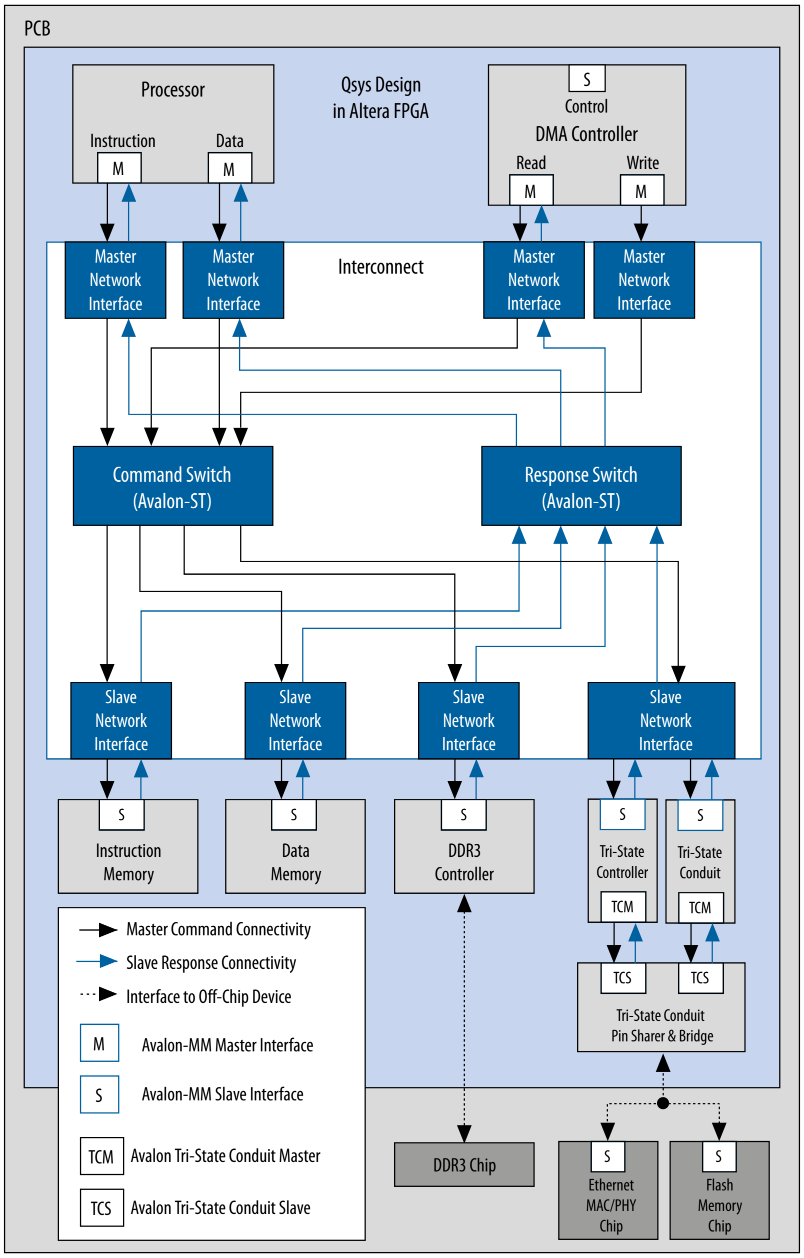
Qsys interconnect supports the following implementation scenarios:

- Any number of components with master and slave interfaces. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- Masters and slaves of different data widths.
- Masters and slaves operating in different clock domains.
- IP Components with different interface properties and signals. Qsys adapts the component interfaces so that interfaces with the following differences can be connected:
  - Avalon and AXI interfaces that use active-high and active-low signaling. AXI signals are active high, except for the reset signal.
  - Interfaces with different burst characteristics.
  - Interfaces with different latencies.
  - Interfaces with different data widths.
  - Interfaces with different optional interface signals.

  Note: AXI3/4 to AXI3/4 interface connections declare a fixed set of signals with variable latency. As a result, there is no need for adapting between active-low and active-high signaling, burst characteristics, different latencies, or port signatures. Some adaptation may be necessary between Avalon interfaces.

Figure 7-1: Qsys interconnect for an Avalon-MM System with Multiple Masters

In this example, there are two components mastering the system, a processor and a DMA controller, each with two master interfaces. The masters connect through the Qsys interconnect to several slaves in the Qsys system. The dark blue blocks represent interconnect components. The dark grey boxes indicate items outside of the Qsys system and the Quartus II software design, and show how component interfaces can be exported and connected to external devices.



Qsys Packet Format

The Qsys packet format supports Avalon, AXI, and APB transactions. Memory-mapped transactions between masters and slaves are encapsulated in Qsys packets. For Avalon systems without AXI or APB interfaces, some fields are ignored or removed.

Qsys Packet Format

Table 7-1: Qsys Packet Format for Memory-Mapped Master and Slave Interfaces

The fields of the Qsys packet format are of variable length to minimize resource usage. However, if the majority of components in a design have a single data width, for example 32-bits, and a single component has a data width of 64-bits, Qsys inserts a width adapter to accommodate 64-bit transfers.

Command	Description
`Address`	Specifies the byte address for the lowest byte in the current cycle. There are no restrictions on address alignment.
`Size`	Encodes the run-time size of the transaction. In conjunction with address, this field describes the segment of the payload that contains valid data for a beat within the packet.
`Address Sideband`	Carries "address" sideband signals. The interconnect passes this field from master to slave. This field is valid for each beat in a packet, even though it is only produced and consumed by an address cycle. Up to 8-bit sideband signals are supported for both read and write address channels.
`Cache`	Carries the AXI cache signals.
`Transaction (Exclusive)`	Indicates whether the transaction has exclusive access.
`Transaction (Posted)`	Used to indicate non-posted writes (writes that require responses).
`Data`	For command packets, carries the data to be written. For read response packets, carries the data that has been read.

Command	Description
`Byteenable`	Specifies which symbols are valid. AXI can issue or accept any byteenable pattern. For compatibility with Avalon, Altera recommends that you use the following legal values for 32-bit data transactions between Avalon masters and slaves: • **1111**—Writes full 32 bits • **0011**—Writes lower 2 bytes • **1100**—Writes upper 2 bytes • **0001**—Writes byte 0 only • **0010**—Writes byte 1 only • **0100**—Writes byte 2 only • **1000**—Writes byte 3 only
`Source_ID`	The ID of the master or slave that initiated the command or response.
`Destination_ID`	The ID of the master or slave to which the command or response is directed.
`Response`	Carries the AXI response signals.
`Thread ID`	Carries the AXI transaction ID values.
`Byte count`	The number of bytes remaining in the transaction, including this beat. Number of bytes requested by the packet.

Command	Description
`Burstwrap`	The burstwrap value specifies the wrapping behavior of the current burst. The burstwrap value is of the form $2^{<n>}$ -1. The following types are defined: • Variable wrap–Variable wrap bursts can wrap at any integer power of 2 value. When the burst reaches the wrap boundary, it wraps back to the previous burst boundary so that only the low order bits are used for addressing. For example, a burst starting at address 0x1C, with a burst wrap boundary of 32 bytes and a burst size of 20 bytes, would write to addresses 0x1C, 0x0, 0x4, 0x8, and 0xC. • For a burst wrap boundary of size <*m*>, `Burstwrap` = <*m*> - 1, or for this case `Burstwrap` = (32 - 1) = 31 which is 2^5 -1. • For AXI masters, the burstwrap boundary value (m) is based on the different `AXBURST`: • Burstwrap set to all 1's. For example, for a 6-bit burstwrap, burstwrap is `6'b111111`. • For `WRAP` bursts, burstwrap = AXLEN * size – 1. • For `FIXED` bursts, burstwrap = size – 1. • Sequential bursts increment the address for each transfer in the burst. For sequential bursts, the `Burstwrap` field is set to all 1s. For example, with a 6-bit `Burstwrap` field, the value for a sequential burst is 6'b111111 or 63, which is 2^6 - 1. For Avalon masters, Qsys adaptation logic sets a hardwired value for the burstwrap field, according the declared master burst properties. For example, for a master that declares sequential bursting, the burstwrap field is set to ones. Similarly, masters that declare burst have their burstwrap field set to the appropriate constant value. AXI masters choose their burst type at run-time, depending on the value of the `AW` or `ARBURST` signal. The interconnect calculates the burstwrap value at run-time for AXI masters.
`Protection`	Access level protection. When the lowest bit is 0, the packet has normal access. When the lowest bit is 1, the packet has privileged access. For Avalon-MM interfaces, this field maps directly to the privileged access signal, which allows an memory-mapped master to write to an on-chip memory ROM instance. The other bits in this field support AXI secure accesses and uses the same encoding, as described in the AXI specification.
`QoS`	QoS (Quality of Service Signaling) is a 4-bit field that is part of the AXI4 interface that carries QoS information for the packet from the AXI master to the AXI slave. Transactions from AXI3 and Avalon masters have the default value `4'b0000`, that indicates that they are not participating in the QoS scheme. QoS values are dropped for slaves that do not support QoS.

Command	Description
`Data sideband`	Carries data sideband signals for the packet. On a write command, the data sideband directly maps to `WUSER`. On a read response, the data sideband directly maps to `RUSER`. On a write response, the data sideband directly maps to `BUSER`.

Transaction Types for Memory-Mapped Interfaces

Table 7-2: Transaction Types for Memory-Mapped Interfaces

The table below describes the information that each bit transports in the packet format's transaction field.

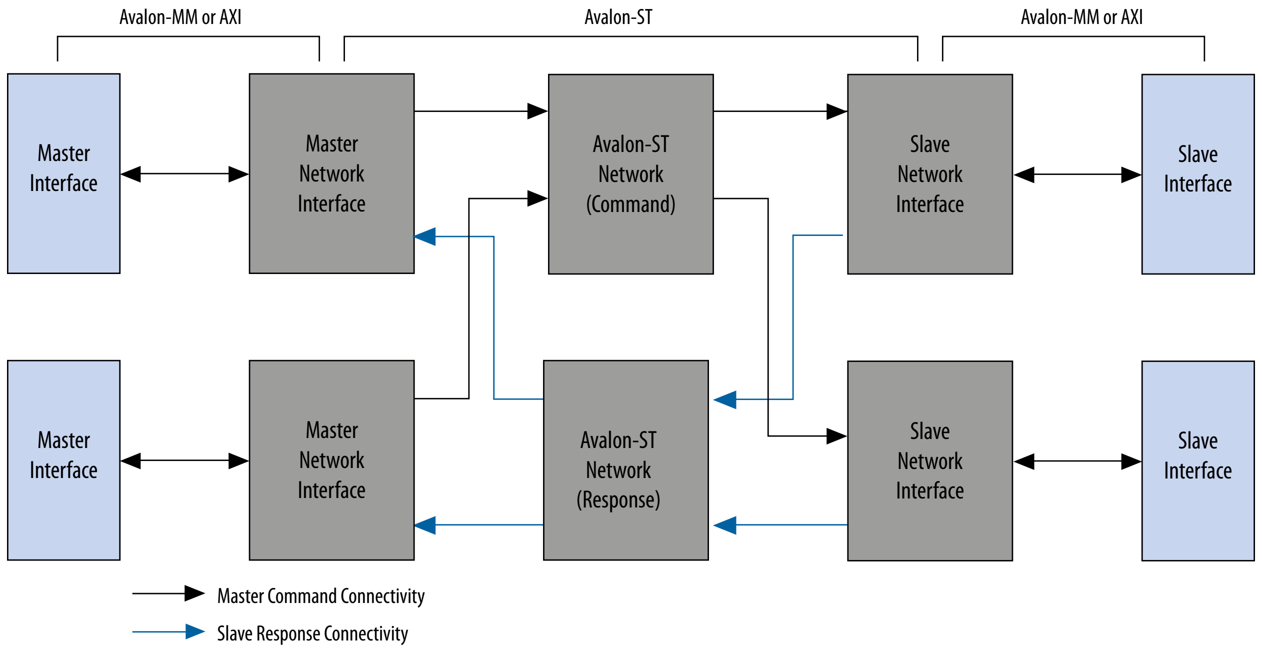
Bit	Name	Definition
0	`PKT_TRANS_READ`	When asserted, indicates a read transaction.
1	`PKT_TRANS_COMPRESSED_READ`	For read transactions, specifies whether or not the read command can be expressed in a single cycle, that is whether or not it has all `byteenables` asserted on every cycle.
2	`PKT_TRANS_WRITE`	When asserted, indicates a write transaction.
3	`PKT_TRANS_POSTED`	When asserted, no response is required.
4	`PKT_TRANS_LOCK`	When asserted, indicates arbitration is locked. Applies to write packets.

Qsys Transformations

The memory-mapped master and slave components connect to network interface modules that encapsulate the transaction in Avalon-ST packets. The memory-mapped interfaces have no information about the encapsulation or the function of the layer transporting the packets. The interfaces operate in accordance with memory-mapped protocol and use the read and write signals and transfers.

Figure 7-2: Transformation when Generating a System with Memory-Mapped and Slave Components

Qsys components that implement the blocks appear shaded.



Related Information

- **Master Network Interfaces** on page 7-11
- **Slave Network Interfaces** on page 7-13

Interconnect Domains

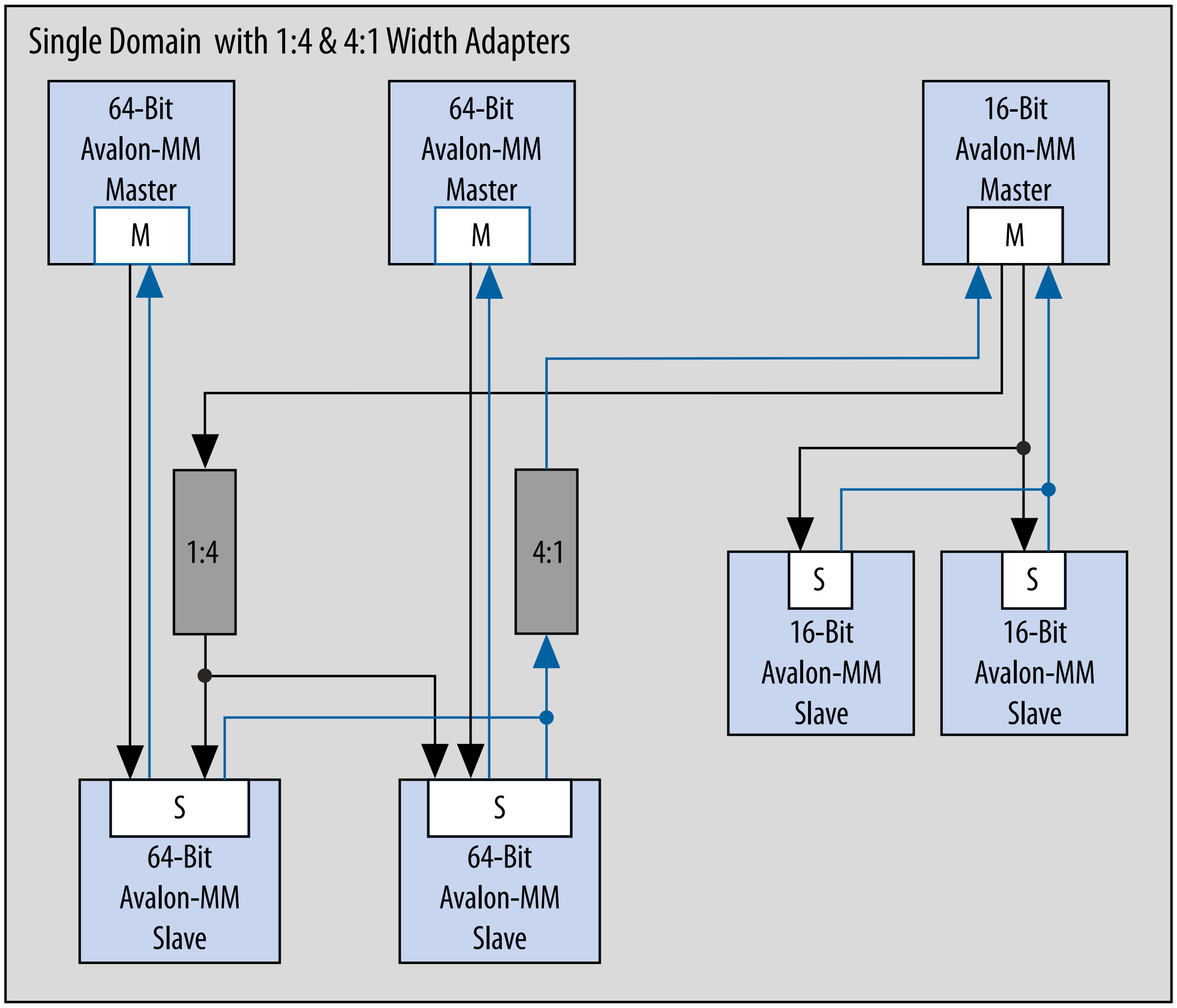
An interconnect domain is a group of connected memory-mapped masters and slaves that share the same interconnect. The components in a single interconnect domain share the same packet format.

Using One Domain with Width Adaptation

When one of the masters in a system connects to all of the slaves, Qsys creates a single domain with two packet formats: one with 64-bit data, and one with 16-bit data. A width adapter manages accesses between the 16-bit master and 64-bit slaves.

Figure 7-3: One Domain with 1:4 and 4:1 Width Adapters

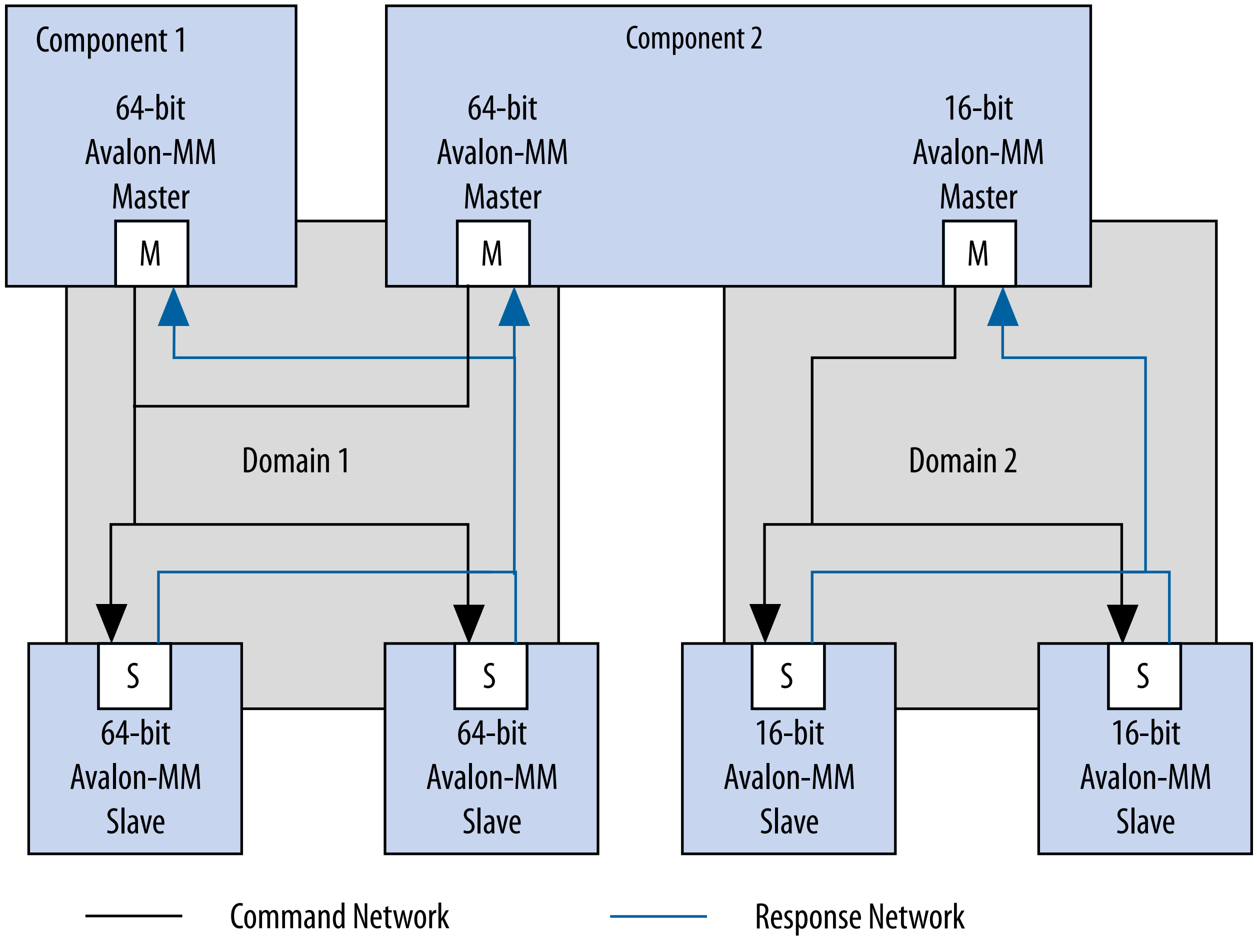
In this system example, there are two 64-bit masters that access two 64-bit slaves. It also includes one 16-bit master, that accesses two 16-bit slaves and two 64-bit slaves. The 16-bit Avalon master connects through a 1:4 adapter, then a 4:1 adapter to reach its 16-bit slaves.



Using Two Separate Domains

Figure 7-4: Two Separate Domains

In this system example, Qsys uses two separate domains. The first domain includes two 64-bit masters connected to two 64-bit slaves. A second domain includes one 16-bit master connected to two 16-bit slaves. Because the interfaces in Domain 1 and Domain 2 do not share any connections, Qsys can optimize the packet format for the two separate domains. In this example, the first domain uses a 64-bit data width and the second domain uses 16-bit data.



Master Network Interfaces

Figure 7-5: Avalon-MM Master Network Interface

Avalon network interfaces drive default values for the `QoS` and `BUSER`, `WUSER`, and `RUSER` packet fields in the master agent, and drop the packet fields in the slave agent.

Note: The `response` signal from the Limiter to the Agent is optional.

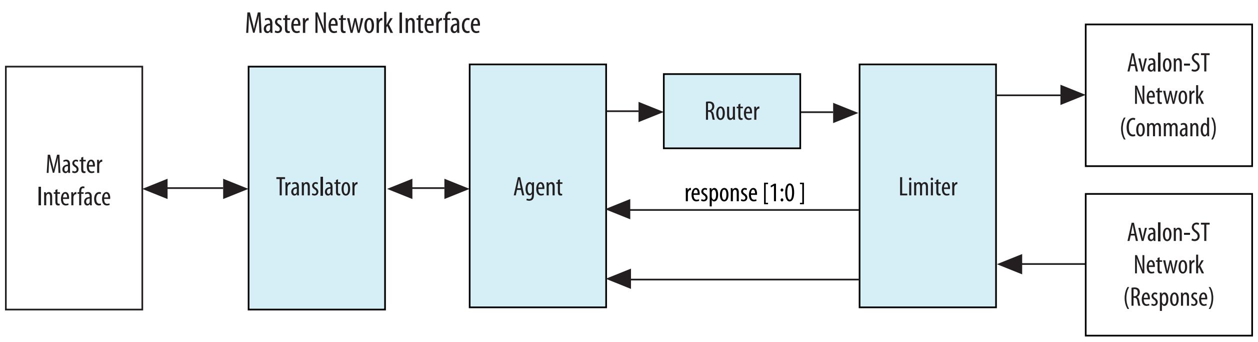
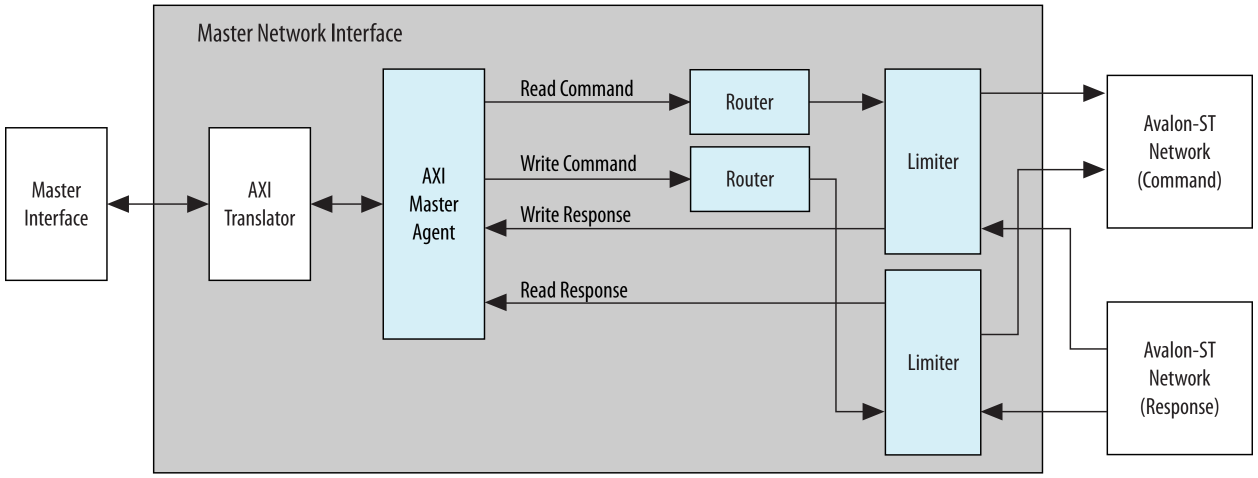


Figure 7-6: AXI Master Network Interface

An AXI4 master supports `INCR` bursts up to 256 beats, QoS signals, and data sideband signals.



Note: For a complete definition of the optional read response signal, refer to *Avalon Memory-Mapped Interface Signal Types* in the *Avalon Interface Specifications*.

Related Information

- **Read and Write Responses** on page 7-27
- **Avalon Interface Specifications**
- **Creating a System with Qsys** on page 5-1

Avalon-MM Master Agent

The Avalon-MM Master Agent translates Avalon-MM master transactions into Qsys command packets and translates the Qsys Avalon-MM slave response packets into Avalon-MM responses.

Avalon-MM Master Translator

The Avalon-MM Master Translator interfaces with an Avalon-MM master component and converts the Avalon-MM master interface to a simpler representation for use in Qsys.

The Avalon-MM Master translator performs the following functions:

- Translates active-low signaling to active-high signaling
- Inserts wait states to prevent an Avalon-MM master from reading invalid data
- Translates word and symbol addresses
- Translates word and symbol burst counts
- Manages re-timing and re-sequencing bursts
- Removes unnecessary address bits

AXI Master Agent

An AXI Master Agent accepts AXI commands and produces Qsys command packets. It also accepts Qsys response packets and converts those into AXI responses. This component has separate packet channels for read commands, write commands, read responses, and write responses. Avalon master agent drives the `QoS` and BUSER, WUSER, and RUSER packet fields with default values `AXQO` and `b0000`, respectively.

Note: For signal descriptions, refer to *Qsys Packet Format*.

Related Information

Qsys Packet Format on page 7-4

AXI Translator

AXI4 allows some signals to be omitted from interfaces. The translator bridges between these "incomplete" AXI4 interfaces and the "complete" AXI4 interface on the network interfaces.

The AXI translator is inserted for both AXI4 masters and slaves and performs the following functions:

- Matches ID widths between the master and slave in 1x1 systems.
- Drives default values as defined in the *AMBA Protocol Specifications* for missing signals.
- Performs lock transaction bit conversion when an AXI3 master connects to an AXI4 slave in 1x1 systems.

Related Information

AMBA Protocol Specifications

APB Master Agent

An APB master agent accepts APB commands and produces or generates Qsys command packets. It also converts Qsys response packets to APB responses.

APB Slave Agent

An APB slave agent issues resulting transaction to the APB interface. It also accepts creates Qsys response packets.

APB Translator

An APB peripheral does not require `pslverr` signals to support additional signals for the APB debug interface.

The APB translator is inserted for both the master and slave and performs the following functions:

- Sets the response value default to `OKAY` if the APB slave does not have a `pslverr` signal.
- Turns on or off additional signals between the APB debug interface, which is used with HPS (Altera SoC's Hard Processor System).

AHB Slave Agent

The Qsys interconnect supports non-bursting Advanced High-performance Bus (AHB) slave interfaces.

Memory-Mapped Router

The Memory-Mapped Router routes command packets from the master to the slave, and response packets from the slave to the master. For master command packets, the router uses the address to set the `Destination_ID` and Avalon-ST channel. For the slave response packet, the router uses the `Destination_ID` to set the Avalon-ST channel. The demultiplexers use the Avalon-ST channel to route the packet to the correct destination.

Memory-Mapped Traffic Limiter

The Memory-Mapped Traffic Limiter ensures the responses arrive in order. It prevents any command from being sent if the response could conflict with the response for a command that has already been issued. By guaranteeing in-order responses, the Traffic Limiter simplifies the response network.

Slave Network Interfaces

Figure 7-7: Avalon-MM Slave Network Interface

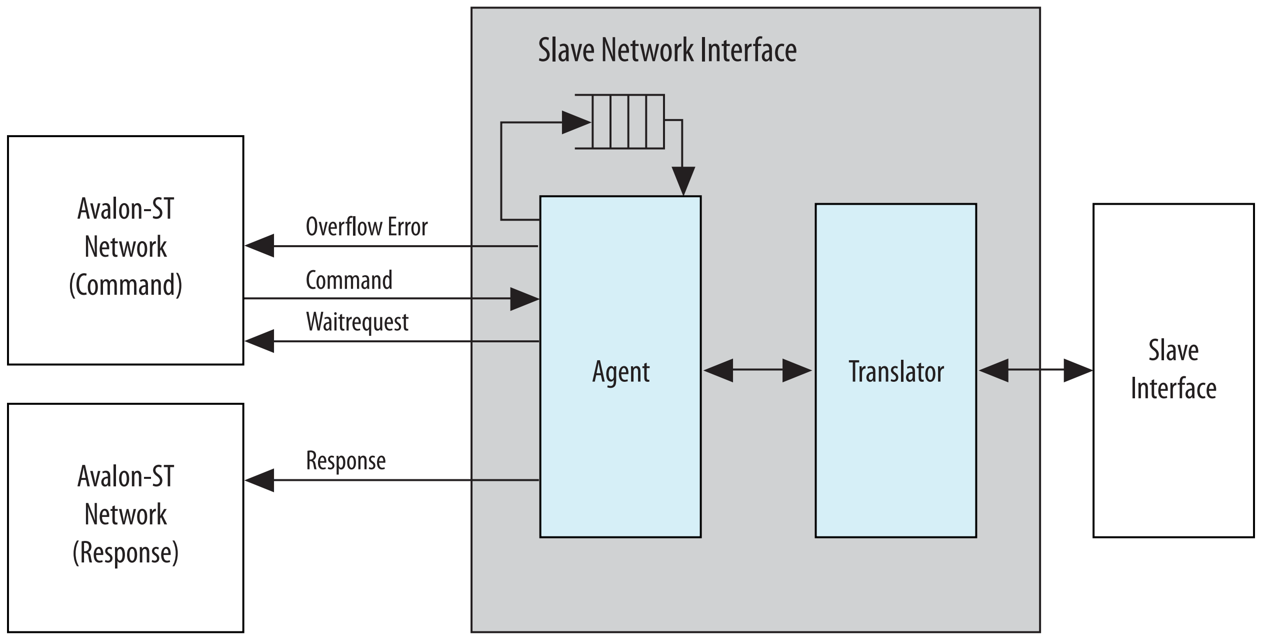
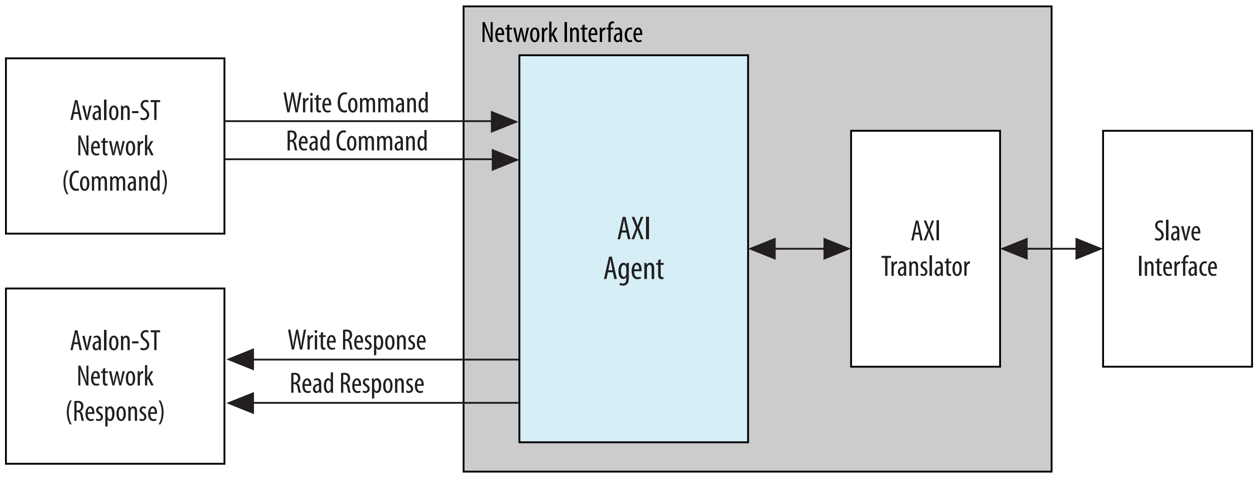


Figure 7-8: AXI Slave Network Interface

An AXI4 slave supports up to 256 beat `INCR` bursts, QoS signals, and data sideband signals.



Avalon-MM Slave Translator

The Avalon-MM Slave Translator interfaces to an Avalon-MM slave component as the *Avalon-MM Slave Network Interface* figure illustrates. It converts the Avalon-MM slave interface to a simplified representation that the Qsys network can use.

An Avalon-MM Merlin Slave Translator performs the following functions:

- Drives the `beginbursttransfer` and `byteenable` signals.
- Supports Avalon-MM slaves that operate using fixed timing and or slaves that use the `readdatavalid` signal to identify valid data.
- Translates the `read`, `write`, and `chipselect` signals into the representation that the Avalon-ST slave response network uses.
- Converts active low signals to active high signals.
- Translates word and symbol addresses and burstcounts.
- Handles burstcount timing and sequencing.
- Removes unnecessary address bits.

Related Information

Slave Network Interfaces on page 7-13

AXI Translator

AXI4 allows some signals to be omitted from interfaces. The translator bridges between these “incomplete” AXI4 interfaces and the “complete” AXI4 interface on the network interfaces.

The AXI translator is inserted for both AXI4 master and slave, and performs the following functions:

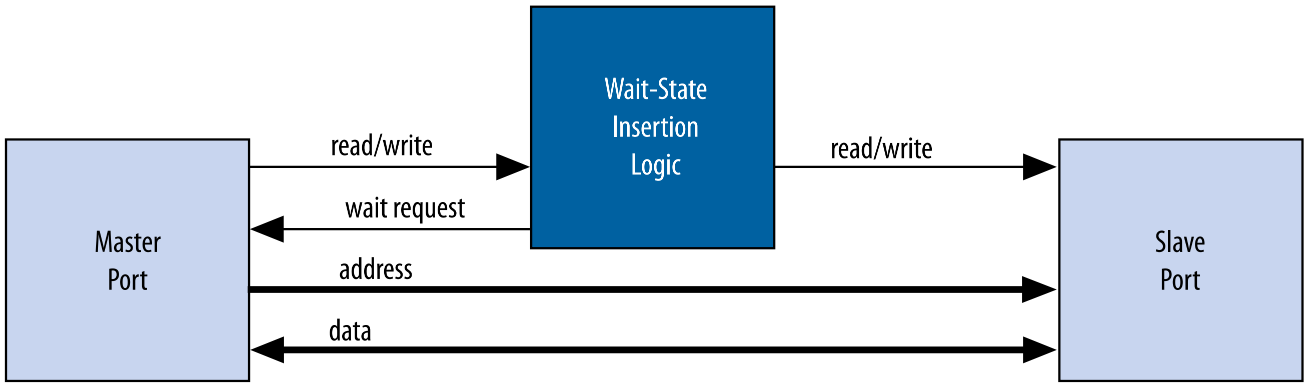
- Matches ID widths between master and slave in 1x1 systems.
- Drives default values as defined in the *AMBA Protocol Specifications* for missing signals.
- Performs lock transaction bit conversion when an AXI3 master connects to an AXI4 slave in 1x1 systems.

Wait State Insertion

Wait states extend the duration of a transfer by one or more cycles. Wait state insertion logic accommodates the timing needs of each slave, and causes the master to wait until the slave can proceed. Qsys interconnect inserts wait states into a transfer when the target slave cannot respond in a single clock cycle, as well as in cases when slave `read` and `write` signals have setup or hold time requirements.

Figure 7-9: Wait State Insertion Logic for One Master and One Slave

Wait state insertion logic is a small finate-state machine that translates control signal sequencing between the slave side and the master side. Qsys interconnect can force a master to wait for the wait state needs of a slave. For example, arbitration logic in a multi-master system. Qsys generates wait state insertion logic based on the properties of all slaves in the system.



Avalon-MM Slave Agent

The Avalon-MM Slave Agent accepts command packets and issues the resulting transactions to the Avalon interface. For pipelined slaves, an Avalon-ST FIFO stores information about pending transactions. The size of this FIFO is the maximum number of pending responses that you specify when creating the slave component. The Avalon-MM Slave Agent also `backpressures` the Avalon-MM master command interface when the FIFO is full if the slave component includes the `waitrequest` signal.

AXI Slave Agent

An AXI Slave Agent works similar to a master agent in reverse. The AXI slave Agent accepts Qsys command packets to create AXI commands, and accepts AXI responses to create Qsys response packets. This component has separate packet channels for read commands, write commands, read responses, and write responses.

Arbitration

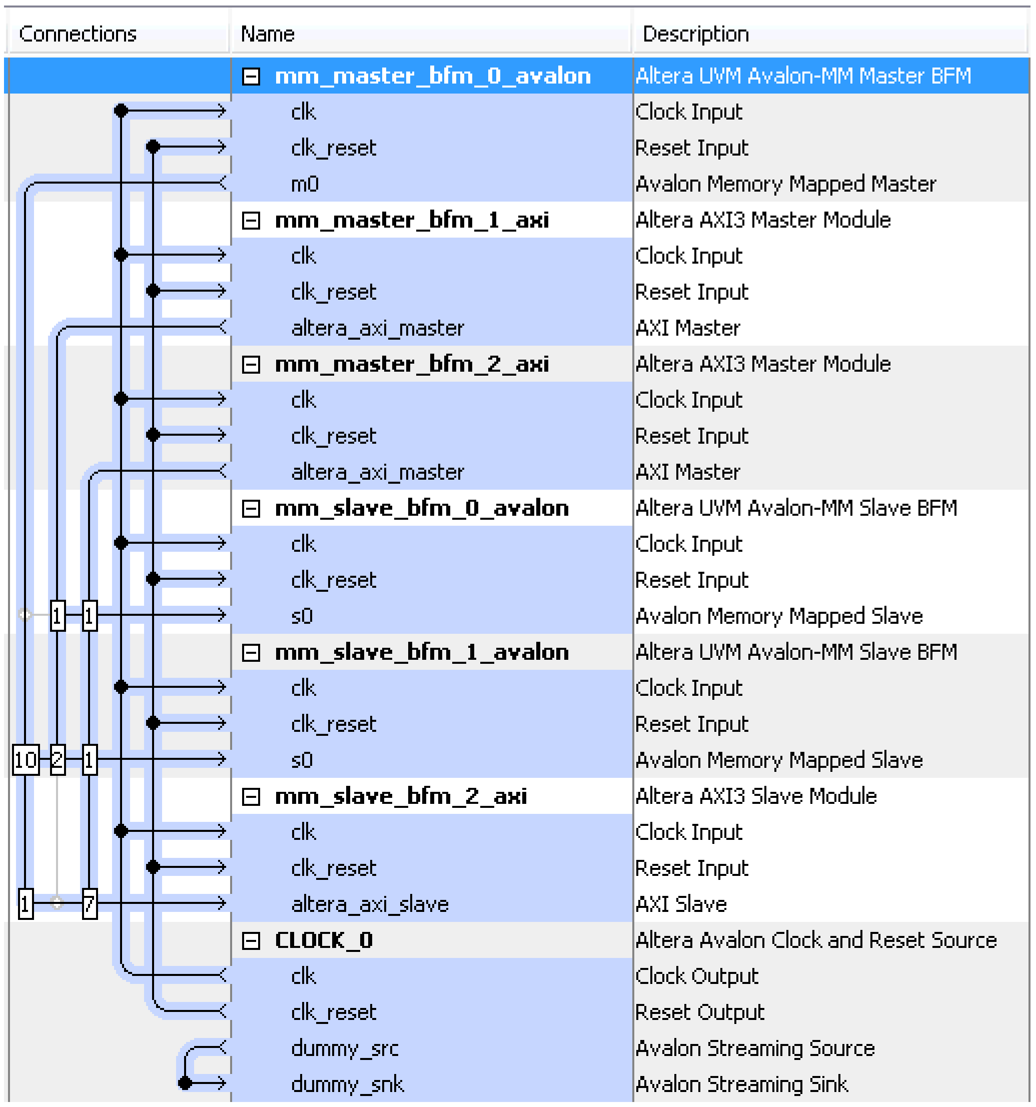
When multiple masters contend for access to a slave, Qsys automatically inserts arbitration logic, which grants access in fairness-based, round-robin order. You can alternatively choose to designate a slave as a fixed priority arbitration slave, and then manually assign priorities in the Qsys GUI.

Round-Robin Arbitration

When multiple masters contend for access to a slave, Qsys automatically inserts arbitration logic which grants access in fairness-based, round-robin order.

In a fairness-based arbitration protocol, each master has an integer value of transfer *shares* with respect to a slave. One share represents permission to perform one transfer. The default arbitration scheme is equal share round-robin that grants equal, sequential access to all requesting masters. You can change the arbitration scheme to weighted round-robin by specifying a relative number of arbitration shares to the masters that access a particular slave. AXI slaves have separate arbitration for their independent read and write channels, and the **Arbitration Shares** setting affects both the read and write arbitration. To display arbitration settings, right-click an instance on the **System Contents** tab, and then click **Show Arbitration Shares**.

Figure 7-10: Arbitration Shares in the Connections Column



Connections	Name	Description
	⊟ mm_master_bfm_0_avalon	Altera UVM Avalon-MM Master BFM
	clk	Clock Input
	clk_reset	Reset Input
	m0	Avalon Memory Mapped Master
	⊟ mm_master_bfm_1_axi	Altera AXI3 Master Module
	clk	Clock Input
	clk_reset	Reset Input
	altera_axi_master	AXI Master
	⊟ mm_master_bfm_2_axi	Altera AXI3 Master Module
	clk	Clock Input
	clk_reset	Reset Input
	altera_axi_master	AXI Master
	⊟ mm_slave_bfm_0_avalon	Altera UVM Avalon-MM Slave BFM
	clk	Clock Input
	clk_reset	Reset Input
1 1	s0	Avalon Memory Mapped Slave
	⊟ mm_slave_bfm_1_avalon	Altera UVM Avalon-MM Slave BFM
	clk	Clock Input
	clk_reset	Reset Input
10 2 1	s0	Avalon Memory Mapped Slave
	⊟ mm_slave_bfm_2_axi	Altera AXI3 Slave Module
	clk	Clock Input
	clk_reset	Reset Input
1 7	altera_axi_slave	AXI Slave
	⊟ CLOCK_0	Altera Avalon Clock and Reset Source
	clk	Clock Output
	clk_reset	Reset Output
	dummy_src	Avalon Streaming Source
	dummy_snk	Avalon Streaming Sink

Fairness-Based Shares

In a fairness-based arbitration scheme, each master-to-slave connection provides a transfer share count. This count is a request for the arbiter to grant a specific number of transfers to this master before giving control to a different master. One share represents permission to perform one transfer.

Figure 7-11: Arbitration of Continuous Transfer Requests from Two Masters

Consider a system with two masters connected to a single slave. Master 1 has its arbitration shares set to three, and Master 2 has its arbitration shares set to four. Master 1 and Master 2 continuously attempt to perform back-to-back transfers to the slave. The arbiter grants Master 1 access to the slave for three transfers, and then grants Master 2 access to the slave for four transfers. This cycle repeats indefinitely. The figure below describes the waveform for this scenario.

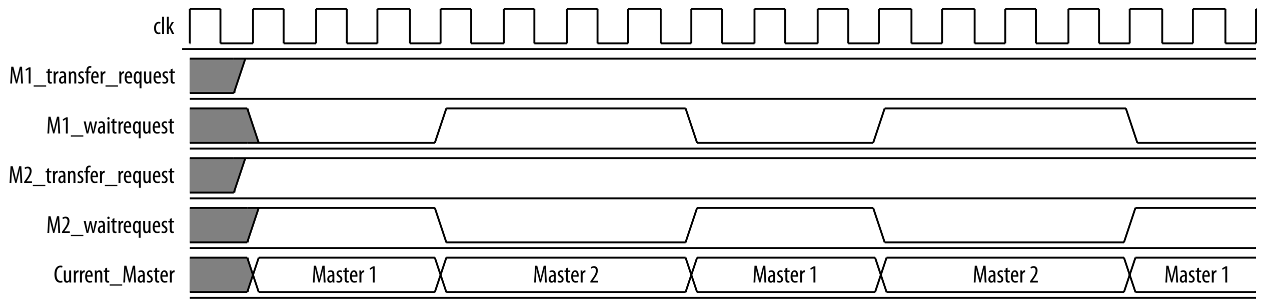
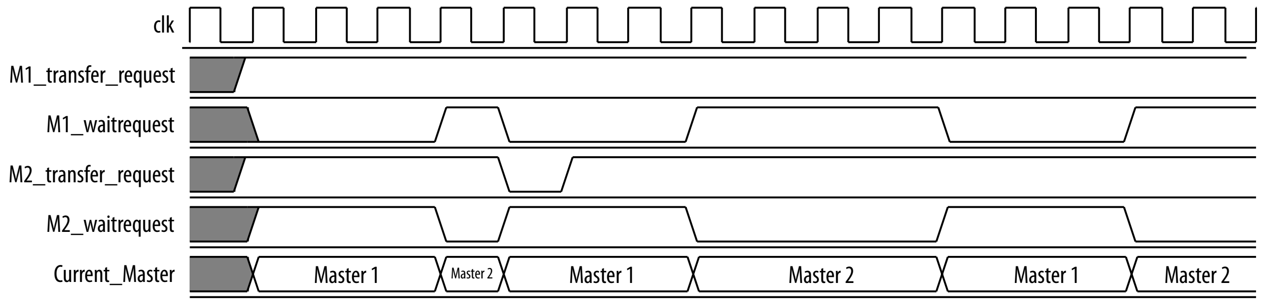


Figure 7-12: Arbitration of Two Masters with a Gap in Transfer Requests

If a master stops requesting transfers before it exhausts its shares, it forfeits all of its remaining shares, and the arbiter grants access to another requesting master. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.



Round-Robin Scheduling

When multiple masters contend for access to a slave, the arbiter grants shares in round-robin order. Qsys includes only requesting masters in the arbitration for each slave transaction.

Fixed Priority Arbitration

Fixed priority arbitration is an alternative arbitration scheme to the default round-robin arbitration scheme.

You can selectively apply fixed priority arbitration to any slave in a Qsys system. You can design Qsys systems where some slaves use the default round-robin arbitration, and other slaves use fixed priority arbitration. Fixed priority arbitration uses a fixed priority algorithm to grant access to a slave amongst its connected masters.

To set up fixed priority arbitration, you must first designate a fixed priority slave in your Qsys system in the **Interconnect Requirements** tab. You can then assign an arbitration priority number for each master connected to a fixed priority slave in the **System Contents** tab, where the highest numeric value receives the highest priority. When multiple masters request access to a fixed priority arbitrated slave, the arbiter gives the master with the highest priority first access to the slave.

For example, when a fixed priority slave receives requests from three masters on the same cycle, the arbiter grants the master with highest assigned priority first access to the slave, and backpreasures the other two masters.

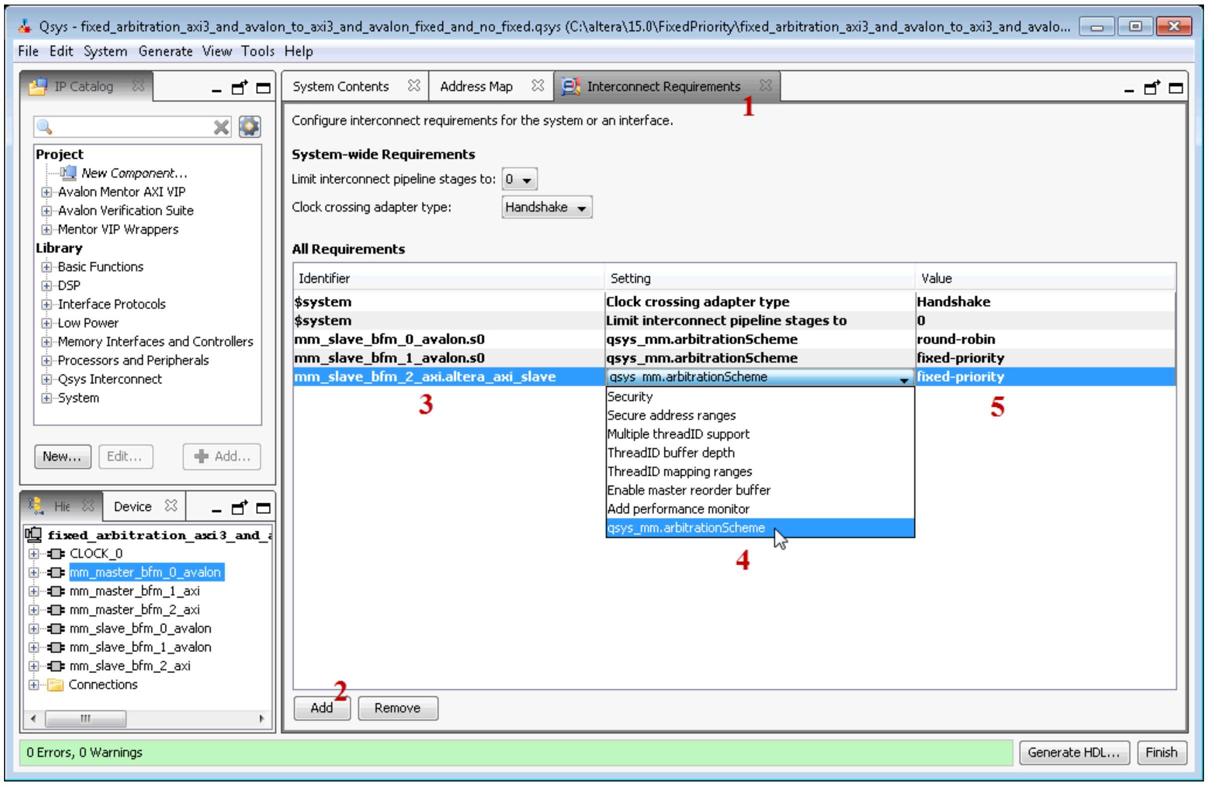
Note: When you connect an AXI master to an Avalon-MM slave designated to use a fixed priority arbitrator, the interconnect instantiates a command-path intermediary round-robin multiplexer in front of the designated slave.

Designate a Qsys Slave to Use Fixed Priority Arbitration

You can designate any slave in your Qsys system to use fixed priority arbitration. You must assign each master connected to a fixed priority slave a numeric priority. The master with the highest higher priority receives first access to the slave. No two masters can have the same priority.

1. In Qsys, navigate to the **Interconnect Requirements** tab.
2. Click **Add** to add a new requirement.
3. In the **Identifier** column, select the slave for fixed priority arbitration.
4. In the **Setting** column, select **qsys mm.arbitrationScheme**.
5. In the **Value** column, select **fixed-priority**.

Figure 7-13: Designate a Slave to Use Fixed Priority Arbitration



Configure interconnect requirements for the system or an interface.

System-wide Requirements

Limit interconnect pipeline stages to: 0

Clock crossing adapter type: Handshake

All Requirements

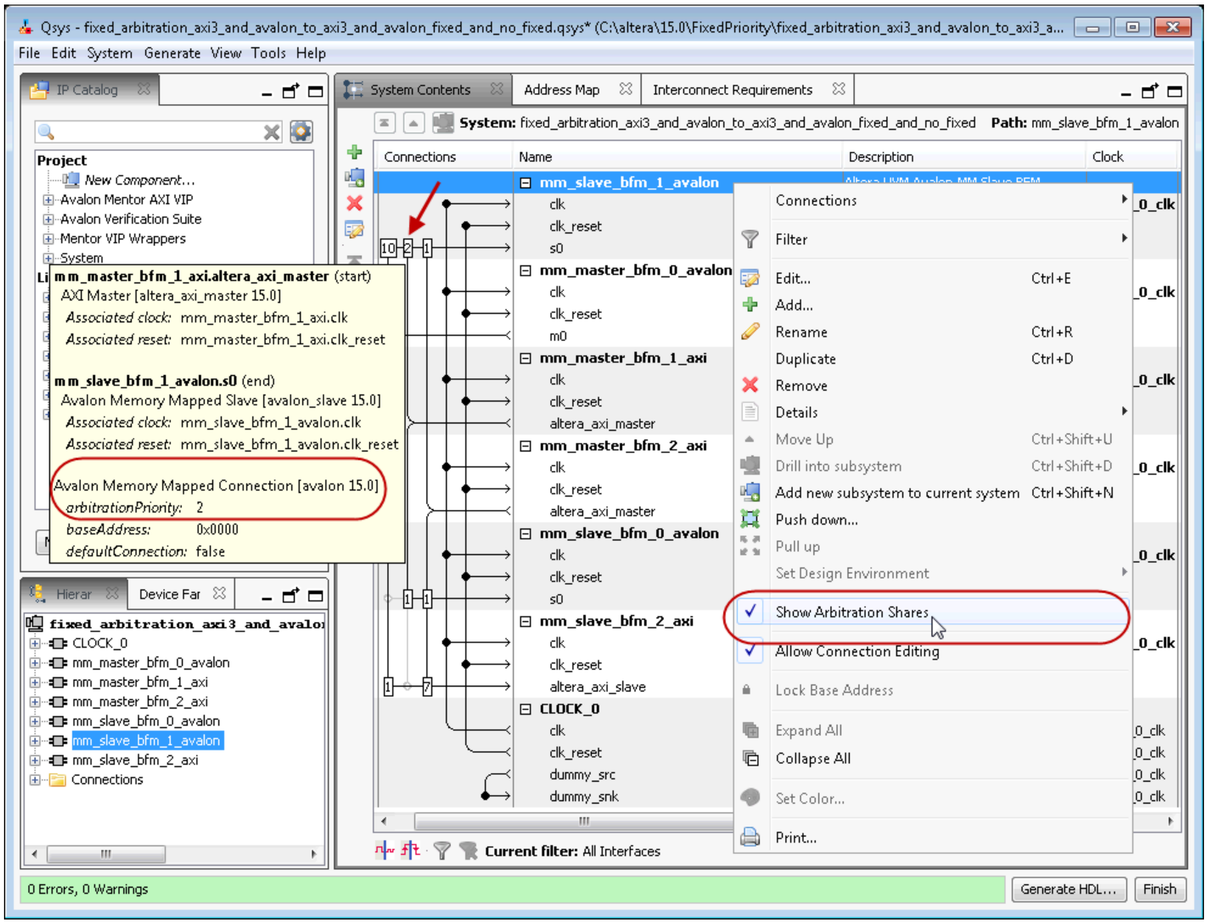
Identifier	Setting	Value
$system	Clock crossing adapter type	Handshake
$system	Limit interconnect pipeline stages to	0
mm_slave_bfm_0_avalon.s0	qsys_mm.arbitrationScheme	round-robin
mm_slave_bfm_1_avalon.s0	qsys_mm.arbitrationScheme	fixed-priority
mm_slave_bfm_2_axi.altera_axi_slave	qsys_mm.arbitrationScheme	fixed-priority

0 Errors, 0 Warnings

6. Navigate to the **System Contents** tab.

7. In the **System Contents** tab, right-click the designated fixed priority slave, and then select **Show Arbitration Shares**.
8. For each master connected to the fixed priory arbitration slave, type a numerical arbitration priority in the box that appears in place of the connection circle.

Figure 7-14: Arbitration Priorities in the Qsys System Contents Tab



Qsys - fixed_arbitration_axi3_and_avalon_to_axi3_and_avalon_fixed_and_no_fixed.qsys* (C:\altera\15.0\FixedPriority\fixed_arbitration_axi3_and_avalon_to_axi3_a...

System: fixed_arbitration_axi3_and_avalon_to_axi3_and_avalon_fixed_and_no_fixed Path: mm_slave_bfm_1_avalon

mm_master_bfm_1_axi.altera_axi_master (start)
AXI Master [altera_axi_master 15.0]
Associated clock: mm_master_bfm_1_axi.clk
Associated reset: mm_master_bfm_1_axi.clk_reset

mm_slave_bfm_1_avalon.s0 (end)
Avalon Memory Mapped Slave [avalon_slave 15.0]
Associated clock: mm_slave_bfm_1_avalon.clk
Associated reset: mm_slave_bfm_1_avalon.clk_reset

Avalon Memory Mapped Connection [avalon 15.0]
arbitrationPriority: 2
baseAddress: 0x0000
defaultConnection: false

Current filter: All Interfaces

0 Errors, 0 Warnings

9. Right click the designated fixed priority slave and uncheck **Show Arbitration Shares** to return to the connection circles.

Fixed Priority Arbitration with AXI Masters and Avalon-MM Slaves

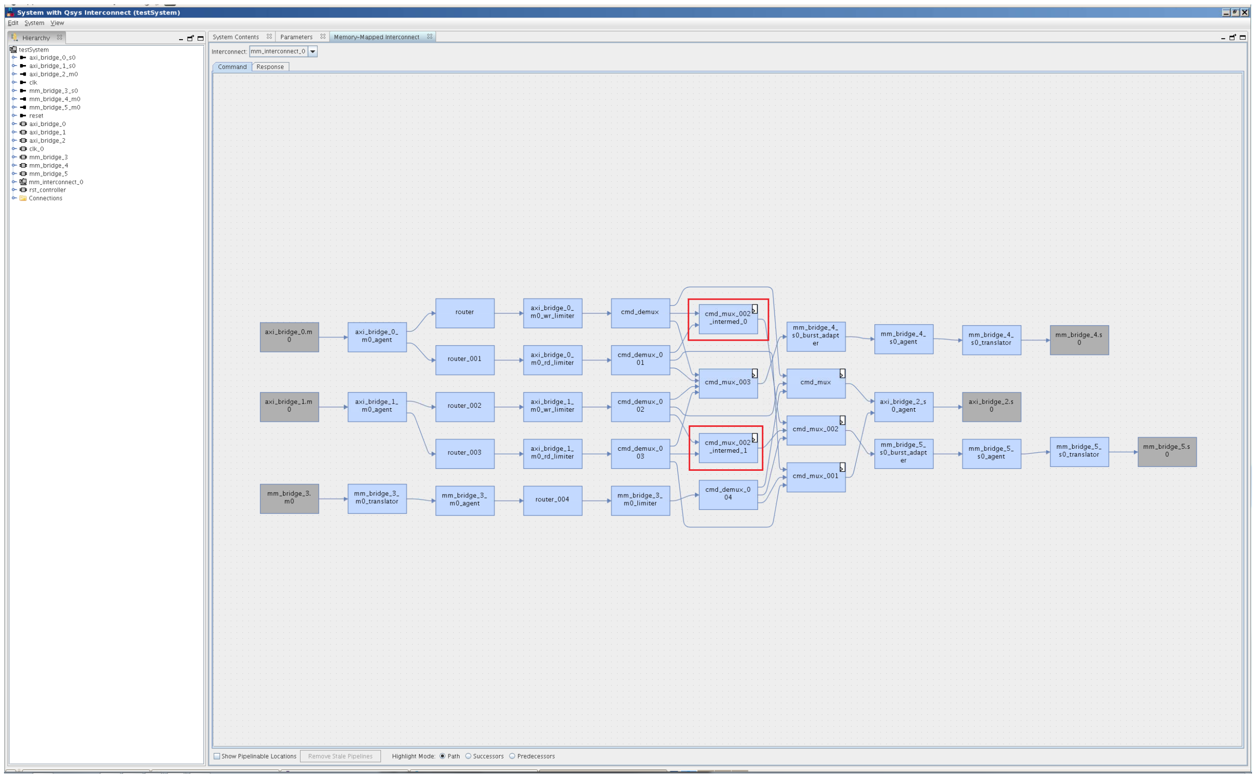
When an AXI master is connected to a designated fixed priority arbitration Avalon-MM slave, Qsys interconnect automatically instantiates an intermediary multiplexer in front of the Avalon-MM slave.

Since AXI masters have separate read and write channels, each channel appears as two separate masters to the Avalon-MM slave. To support fairness between the AXI master's read and write channels, the instantiated round-robin intermediary multiplexer arbitrates between simultaneous read and write commands from the AXI master to the fixed-priority Avalon-MM slave.

When an AXI master is connected to a fixed priority AXI slave, the master's read and write channels are directly connected to the AXI slave's fixed-priority multiplexers. In this case, there is one multiplexer for the read command, and one multiplexer for the write command and therefore an intermediary multiplexer is not required.

The red circles indicate placement of the intermediary multiplexer between the AXI master and Avalon-MM slave due to the separate read and write channels of the AXI master.

Figure 7-15: Intermediary Multiplexer Between AXI Master and Avalon-MM Slave

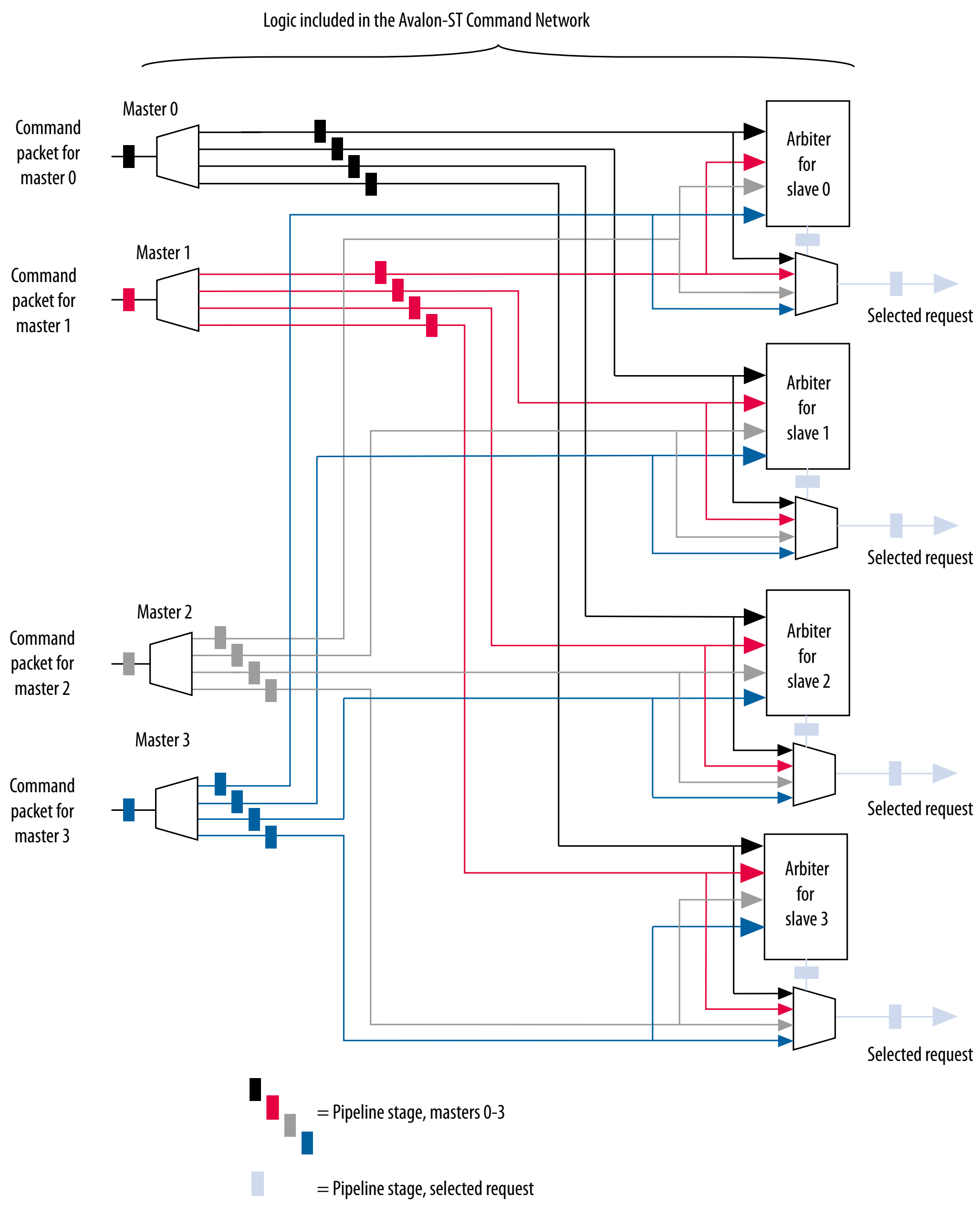


Memory-Mapped Arbiter

The input to the Memory-Mapped Arbiter is the command packet for all masters requesting access to a particular slave. The arbiter outputs the channel number for the selected master. This channel number controls the output of a multiplexer that selects the slave device. The figure below illustrates this logic.

Figure 7-16: Arbitration Logic

In this example, four Avalon-MM masters connect to four Avalon-MM slaves. In each cycle, an arbiter positioned in front of each Avalon-MM slave selects among the requesting Avalon-MM masters.


Logic included in the Avalon-ST Command Network
Master 0
Command packet for master 0
Master 1
Command packet for master 1
Master 2
Command packet for master 2
Master 3
Command packet for master 3
Arbiter for slave 0
Arbiter for slave 1
Arbiter for slave 2
Arbiter for slave 3
Selected request
Selected request
Selected request
Selected request
= Pipeline stage, masters 0-3
= Pipeline stage, selected request

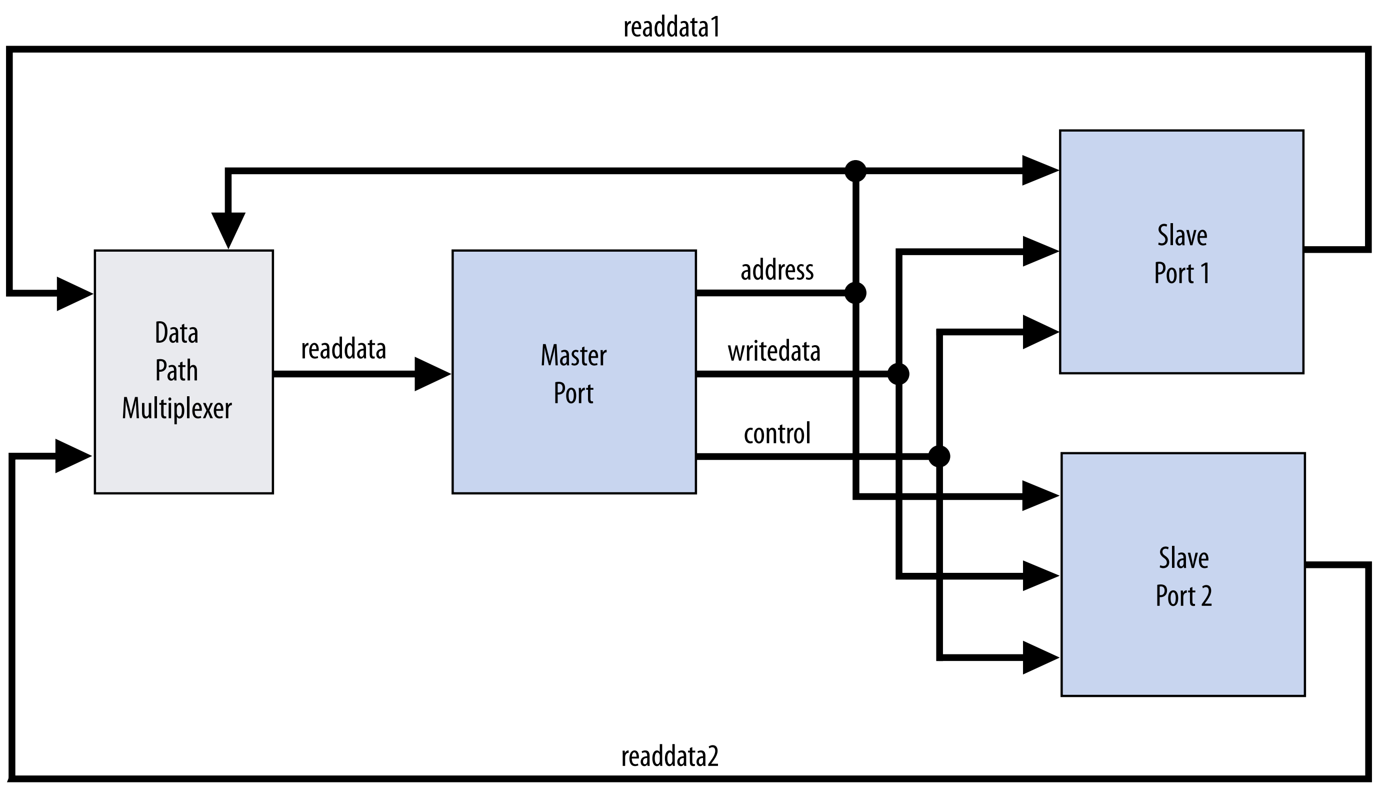
Note: If you specify a **Limit interconnect pipeline stages to** parameter greater than zero, the output of the Arbiter is registered. Registering this output reduces the amount of combinational logic between the master and the interconnect, increasing the f_{MAX} of the system.

Note: You can use the Memory-Mapped Arbiter for both round-robin and fixed priority arbitration.

Datapath Multiplexing Logic

Datapath multiplexing logic drives the `writedata` signal from the granted master to the selected slave, and the `readdata` signal from the selected slave back to the requesting master. Qsys generates separate datapath multiplexing logic for every master in the system (`readdata`), and for every slave in the system (`writedata`). Qsys does not generate multiplexing logic if it is not needed.

Figure 7-17: Datapath Multiplexing Logic for One Master and Two Slaves



Width Adaptation

Qsys width adaptation converts between Avalon memory-mapped master and slaves with different data and byte enable widths, and manages the run-time size requirements of AXI. Width adaptation for AXI to Avalon interfaces is also supported.

Memory-Mapped Width Adapter

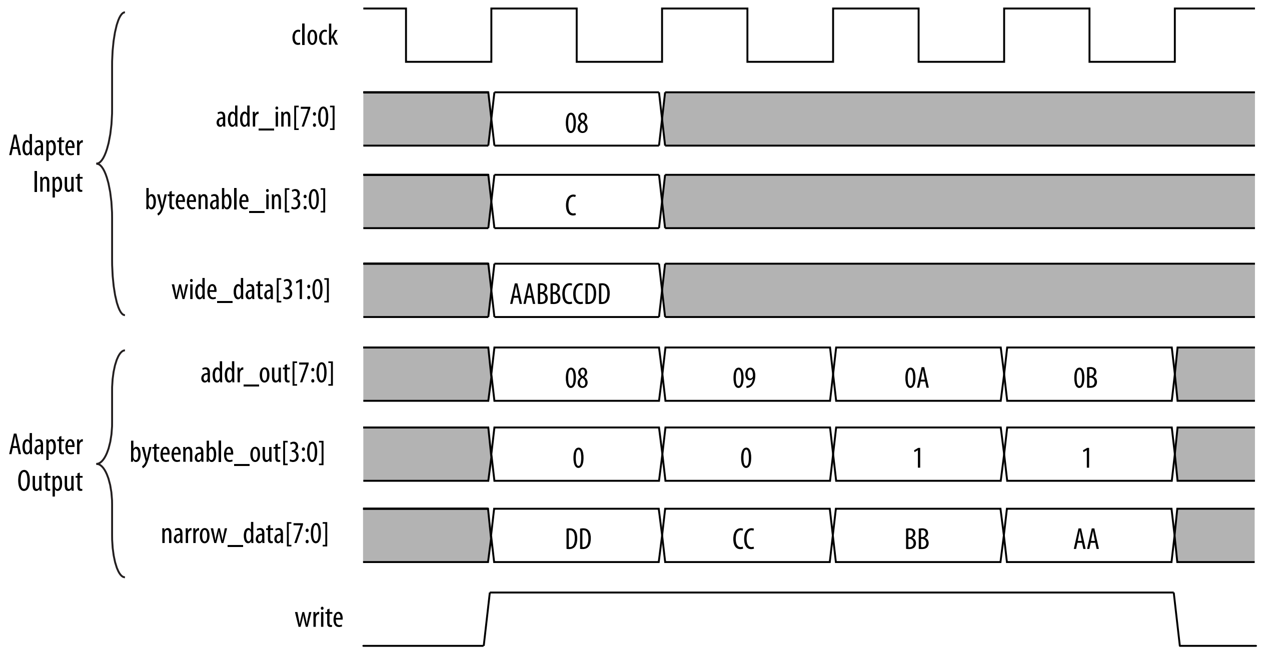
The Memory-Mapped Width Adapter is used in the Avalon-ST domain and operates with information contained in the packet format.

The memory-mapped width adapter accepts packets on its sink interface with one data width and produces output packets on its source interface with a different data width. The ratio of the narrow data width must be a power of two, such as 1:4, 1:8, and 1:16. The ratio of the wider data width to the narrower width must also be a power of two, such as 4:1, 8:1, and 16:1 These output packets may have a different size if the input size exceeds the output data bus width, or if data packing is enabled.

When the width adapter converts from narrow data to wide data, each input beat's data and byte enables are copied to the appropriate segment of the wider output data and byte enables signals.

Figure 7-18: Width Adapter Timing for a 4:1 Adapter

This adapter assumes that the field ordering of the input and output packets is the same, with the only difference being the width of the data and accompanying byte enable fields. When the width adapter converts from wide data to narrow data, the narrower data is transmitted over several beats. The first output beat contains the lowest addressed segment of the input data and byte enables.



AXI Wide-to-Narrow Adaptation

For all cases of AXI wide-to-narrow adaptation, read data is re-packed to match the original size. Responses are merged, with the following error precedence: `DECERR`, `SLVERR`, `OKAY`, and `EXOKAY`.

Table 7-3: AXI Wide-to-Narrow Adaptation (Downsizing)

Burst Type	Behavior
Incrementing	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted to an incrementing burst with a larger length and size equal to the output width. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths. For example, for a 2:1 downsizing ratio, an `INCR9` burst is converted into `INCR16 + INCR2` bursts. This is true if the maximum burstcount a slave can accept is 16, which is the case for AXI3 slaves. Avalon slaves have a maximum burstcount of 64.

Burst Type	Behavior
Wrapping	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted to a wrapping burst with a larger length, with a size equal to the output width. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths; respecting wrap boundaries. For example, for a 2:1 downsizing ratio, a `WRAP16` burst is converted into two or three `INCR` bursts, depending on the address.
Fixed	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted into repeated sequential bursts over the same addresses. For example, for a 2:1 downsizing ratio, a FIXED single burst is converted into an `INCR2` burst.

AXI Narrow-to-Wide Adaptation

Table 7-4: AXI Narrow-to-Wide Adaptation (Upsizing)

Burst Type	Behavior
Incrementing	The burst (and its response) passes through unmodified. Data and write strobes are placed in the correct output segment.
Wrapping	The burst (and its response) passes through unmodified.
Fixed	The burst (and its response) passes through unmodified.

Burst Adapter

Qsys interconnect uses the memory-mapped burst adapter to accommodate the burst capabilities of each interface in the system, including interfaces that do not support burst transfers.

The maximum burst length for each interface is a property of the interface and is independent of other interfaces in the system. Therefore, a particular master may be capable of initiating a burst longer than a slave's maximum supported burst length. In this case, the burst adapter translates the large master burst into smaller bursts, or into individual slave transfers if the slave does not support bursting. Until the master completes the burst, arbiter logic prevents other masters from accessing the target slave. For example, if a master initiates a burst of 16 transfers to a slave with maximum burst length of 8, the burst adapter initiates 2 bursts of length 8 to the slave.

Avalon-MM and AXI burst transactions allow a master uninterrupted access to a slave for a specified number of transfers. The master specifies the number of transfers when it initiates the burst. Once a burst begins between a master and slave, arbiter logic is locked until the burst completes. For burst masters, the length of the burst is the number of cycles that the master has access to the slave, and the selected arbitration shares have no effect.

Note: AXI masters can issue burst types that Avalon cannot accept, for example, fixed bursts. In this case, the burst adapter converts the fixed burst into a sequence of transactions to the same address.

Note: For AXI4 slaves, Qsys allows 256-beat `INCR` bursts. You must ensure that 256-beat narrow-sized `INCR` bursts are shortened to 16-beat narrow-sized INCR bursts for AXI3 slaves.

Avalon-MM masters always issue addresses that are aligned to the size of the transfer. However, when Qsys uses a narrow-to-wide width adaptation, the resulting address may be unaligned. For unaligned addresses, the burst adapter issues the maximum sized bursts with appropriate byte enables. This brings the burst-in-progress up to an aligned slave address. Then, it completes the burst on aligned addresses.

The burst adapter supports variable wrap or sequential burst types to accommodate different properties of memory-mapped masters. Some bursting masters can issue more than one burst type.

Burst adaptation is available for Avalon to Avalon, Avalon to AXI, and AXI to Avalon, and AXI to AXI connections. For information about AXI-to-AXI adaptation, refer to *AXI Wide-to-Narrow Adaptation*

Note: For AXI4 to AXI3 connections, Qsys follows an AXI4 256 burst length to AXI3 16 burst length.

Burst Adapter Implementation Options

Qsys automatically inserts burst adapters into your system depending on your master and slave connections, and properties. You can select burst adapter implementation options on the **Interconnect Requirements** tab.

To access the implementation options, you must select the **Burst adapter implementation** setting for the `$system` identifier.

- **Generic converter (slower, lower area)**—Default. Controls all burst conversions with a single converter that is able to adapt incoming burst types. This results in an adapter that has lower f_{max}, but smaller area.
- **Per-burst-type converter (faster, higher area)**—Controls incoming bursts with a particular converter, depending on the burst type. This results in an adapter that has higher f_{max}, but higher area. This setting is useful when you have AXI masters or slaves and you want a higher f_{max}.

Note: For more information about the **Interconnect Requirements** tab, refer to *Creating a System with Qsys*.

Related Information

- **Creating a System with Qsys** on page 5-1

Burst Adaptation: AXI to Avalon

Table 7-5: Burst Adaptation: AXI to Avalon

Entries specify the behavior when converting between AXI and Avalon burst types.

Burst Type	Behavior
Incrementing	**Sequential Slave** Bursts that exceed `slave_max_burst_length` are converted to multiple sequential bursts of a length less than or equal to the `slave_max_burst_length`. Otherwise, the burst is unconverted. For example, for an Avalon slave with a maximum burst length of 4, an `INCR7` burst is converted to `INCR4 + INCR3`. **Wrapping Slave** Bursts that exceed the `slave_max_burst_length` are converted to multiple sequential bursts of length less than or equal to the `slave_max_burst_length`. Bursts that exceed the wrapping boundary are converted to multiple sequential bursts that respect the slave's wrapping boundary.
Wrapping	**Sequential Slave** A WRAP burst is converted to multiple sequential bursts. The sequential bursts are less than or equal to the `max_burst_length` and respect the transaction's wrapping boundary **Wrapping Slave** If the WRAP transaction's boundary matches the slave's boundary, then the burst passes through. Otherwise, the burst is converted to sequential bursts that respect both the transaction and slave wrap boundaries.
Fixed	Fixed bursts are converted to sequential bursts of length 1 that repeatedly access the same address.
Narrow	All narrow-sized bursts are broken into multiple bursts of length 1.

Burst Adaptation: Avalon to AXI

Table 7-6: Burst Adaptation: Avalon to AXI

Entries specify the behavior when converting between Avalon and AXI burst types.

Burst Type	Definition
Sequential	Bursts of length greater than16 are converted to multiple `INCR` bursts of a length less than or equal to16. Bursts of length less than or equal to16 are not converted.

Burst Type	Definition
Wrapping	Only Avalon masters with `alwaysBurstMaxBurst = true` are supported. The WRAP burst is passed through if the length is less than or equal to16. Otherwise, it is converted to two or more `INCR` bursts that respect the transaction's wrap boundary.
`GENERIC_CONVERTER`	Controls all burst conversions with a single converter that is able to adapt all incoming burst types. This results in an adapter that has smaller area, but lower f_{Max}.

Read and Write Responses

Qsys merges write responses if a write is converted (burst adapted) into multiple bursts. Qsys requires read response merging for a downsized (wide-to-narrow width adapted) read.

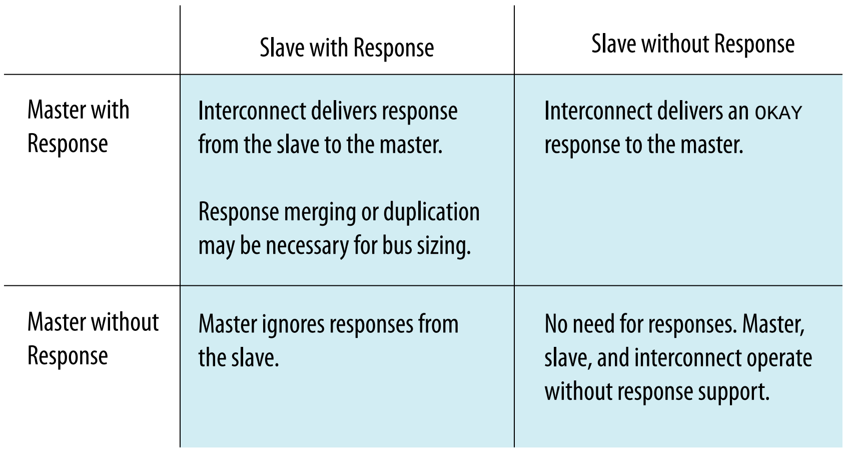
Qsys merges responses based on the following precedence rule:

```
DECERR > SLVERR > OKAY > EXOKAY
```

Adaptation between a master with write responses and a slave without write responses can be costly, especially if there are multiple slaves, or the slave supports bursts. To minimize the cost of logic between slaves, consider placing the slaves that do not have write responses behind a bridge so that the write response adaptation logic cost is only incurred once, at the bridge’s slave interface.

The following table describes what happens when there is a mismatch in response support between the master and slave.

Figure 7-19: Mismatched Master and Slave Response Support



	Slave with Response	Slave without Response
Master with Response	Interconnect delivers response from the slave to the master. Response merging or duplication may be necessary for bus sizing.	Interconnect delivers an OKAY response to the master.
Master without Response	Master ignores responses from the slave.	No need for responses. Master, slave, and interconnect operate without response support.

For the response case where the transaction violates security settings or uses an illegal address, the interconnect routes the transactions to the default slave. For information about Qsys system security and how to specify a default slave, refer to *Creating a System with Qsys*.

Note: Avalon-MM slaves without a `response` signal are not able to notify a connected master that a transaction has not completed successfully. As a result, Qsys interconnect generates an `OKAY` response on behalf of the Avalon-MM slave.

Related Information

- **Master Network Interfaces** on page 7-11
- **Error Correction Coding (ECC) in Qsys Interconnect** on page 7-70
- **Avalon-MM Interface Signal Roles**
- **Interface Properties**

Transaction Order for Avalon-MM Read and Write Responses

For any Avalon-MM slave, commands must be processed in a risk-free manner. Read and write responses are issued following the order in which commands are accepted.

For any Avalon-MM master:

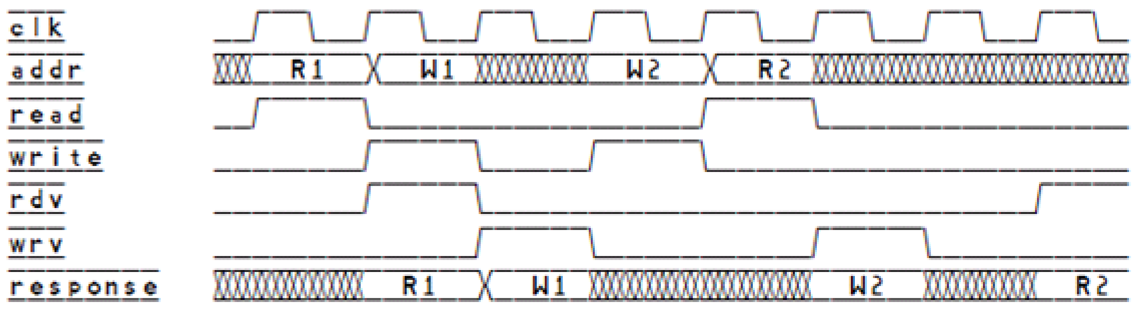
- Commands to the same slave are guaranteed to reach the slave in command issue order, and the slave responds in command issue order.
- Commands to different slaves may reach and be responded to by the slaves in a different order than the order in which the commands were issued. When successful, the slave responds in command issue order.
- Responses (if present) return in command issue order, regardless of whether the read or write commands are directed to the same or different slaves.

Avalon-MM Read and Write Responses Timing Diagram

The following diagram shows command acceptance and command issue order for Avalon-MM read and write responses.

In this example, the `readdatavalid` signal displays as `rdv`, and the `writeresponsevalid` as `wrv`.

Figure 7-20: Avalon-MM Read and Write Responses Timing Diagram



Qsys Address Decoding

Address decoding logic forwards appropriate addresses to each slave.

Address decoding logic simplifies component design in the following ways:

- The interconnect selects a slave whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave addresses are properly aligned to the slave interface.
- Changing the system memory map does not involve manually editing HDL.

Figure 7-21: Address Decoding for One Master and Two Slaves

In this example, Qsys generates separate address decoding logic for each master in a system. The address decoding logic processes the difference between the master address width (`<M>`) and the individual slave address widths (`<S` >) and (`<T` >). The address decoding logic also maps only the necessary master address bits to access words in each slave's address space.

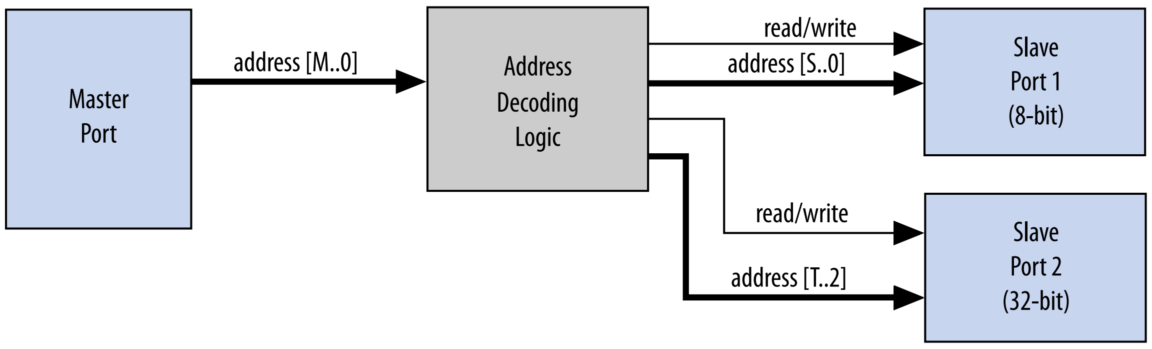
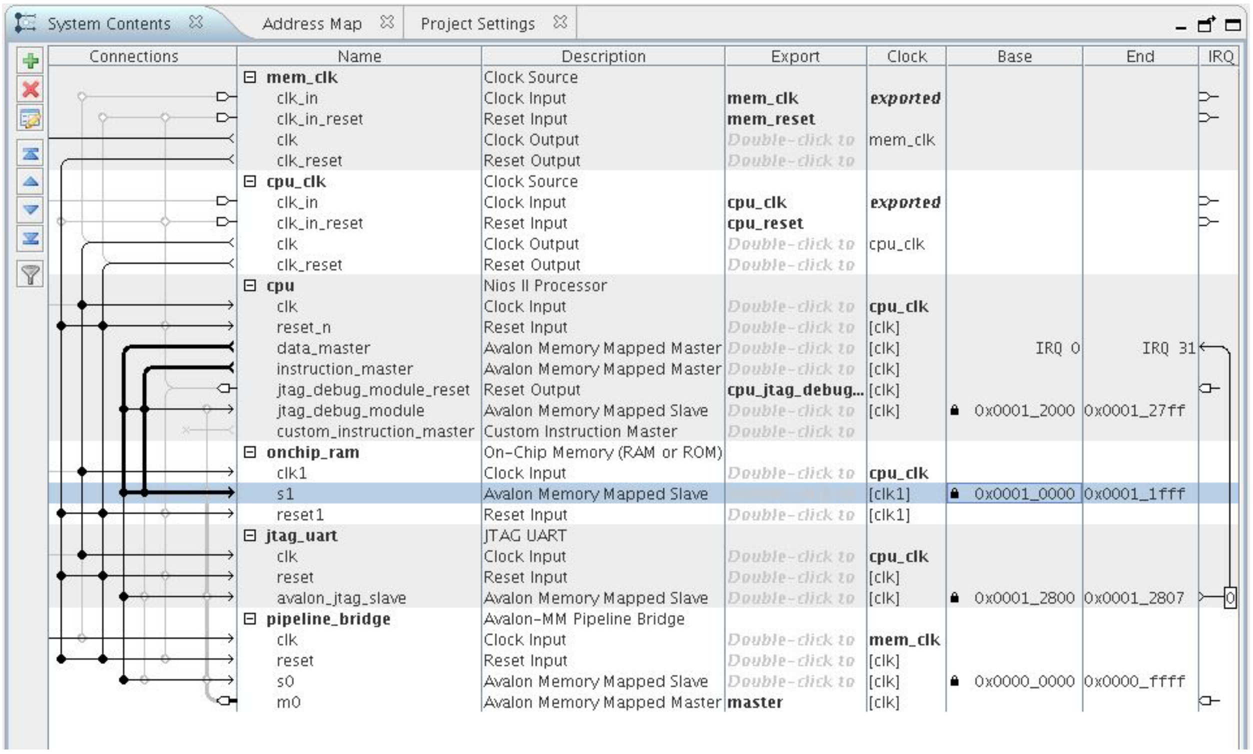


Figure 7-22: Address Decoding Base Settings

Qsys controls the base addresses with the **Base** setting of active components on the **System Contents** tab. The base address of a slave component must be a multiple of the address span of the component. This restriction is part of the Qsys interconnect to allow the address decoding logic to be efficient, and to achieve the best possible f_{MAX}.



System Contents | Address Map | Project Settings

Connections	Name	Description	Export	Clock	Base	End	IRQ
	⊟ **mem_clk**	Clock Source					
	clk_in	Clock Input	**mem_clk**	***exported***			
	clk_in_reset	Reset Input	**mem_reset**				
	clk	Clock Output	Double-click to	mem_clk			
	clk_reset	Reset Output	Double-click to				
	⊟ **cpu_clk**	Clock Source					
	clk_in	Clock Input	**cpu_clk**	***exported***			
	clk_in_reset	Reset Input	**cpu_reset**				
	clk	Clock Output	Double-click to	cpu_clk			
	clk_reset	Reset Output	Double-click to				
	⊟ **cpu**	Nios II Processor					
	clk	Clock Input	Double-click to	**cpu_clk**			
	reset_n	Reset Input	Double-click to	[clk]			
	data_master	Avalon Memory Mapped Master	Double-click to	[clk]	IRQ 0	IRQ 31	
	instruction_master	Avalon Memory Mapped Master	Double-click to	[clk]			
	jtag_debug_module_reset	Reset Output	**cpu_jtag_debug...**	[clk]			
	jtag_debug_module	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0001_2000	0x0001_27ff	
	custom_instruction_master	Custom Instruction Master	Double-click to				
	⊟ **onchip_ram**	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	Double-click to	**cpu_clk**			
	s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0001_0000	0x0001_1fff	
	reset1	Reset Input	Double-click to	[clk1]			
	⊟ **jtag_uart**	JTAG UART					
	clk	Clock Input	Double-click to	**cpu_clk**			
	reset	Reset Input	Double-click to	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0001_2800	0x0001_2807	0
	⊟ **pipeline_bridge**	Avalon-MM Pipeline Bridge					
	clk	Clock Input	Double-click to	**mem_clk**			
	reset	Reset Input	Double-click to	[clk]			
	s0	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0000_0000	0x0000_ffff	
	m0	Avalon Memory Mapped Master	**master**	[clk]			

Avalon Streaming Interfaces

High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. Streaming interfaces can also use memory-mapped connection interfaces to provide an access point for control. In contrast to the memory-mapped interconnect, the Avalon-ST interconnect always creates a point-to-point connection between a single data source and data sink.

Figure 7-23: Memory-Mapped and Avalon-ST Interfaces

In this example, there are the following connection pairs:

- Data source in the Rx Interface transfers data to the data sink in the FIFO.
- Data source in the FIFO transfers data to the `Tx Interface` data sink.

The memory-mapped interface allows a processor to access the data source, FIFO, or data sink to provide system control. If your source and sink interfaces have different formats, for example, a 32-bit source and an 8-bit sink, Qsys automatically inserts the necessary adapters. You can view the adapters on the **System Contents** tab by clicking **System** > **Show System with Qsys Interconnect**.

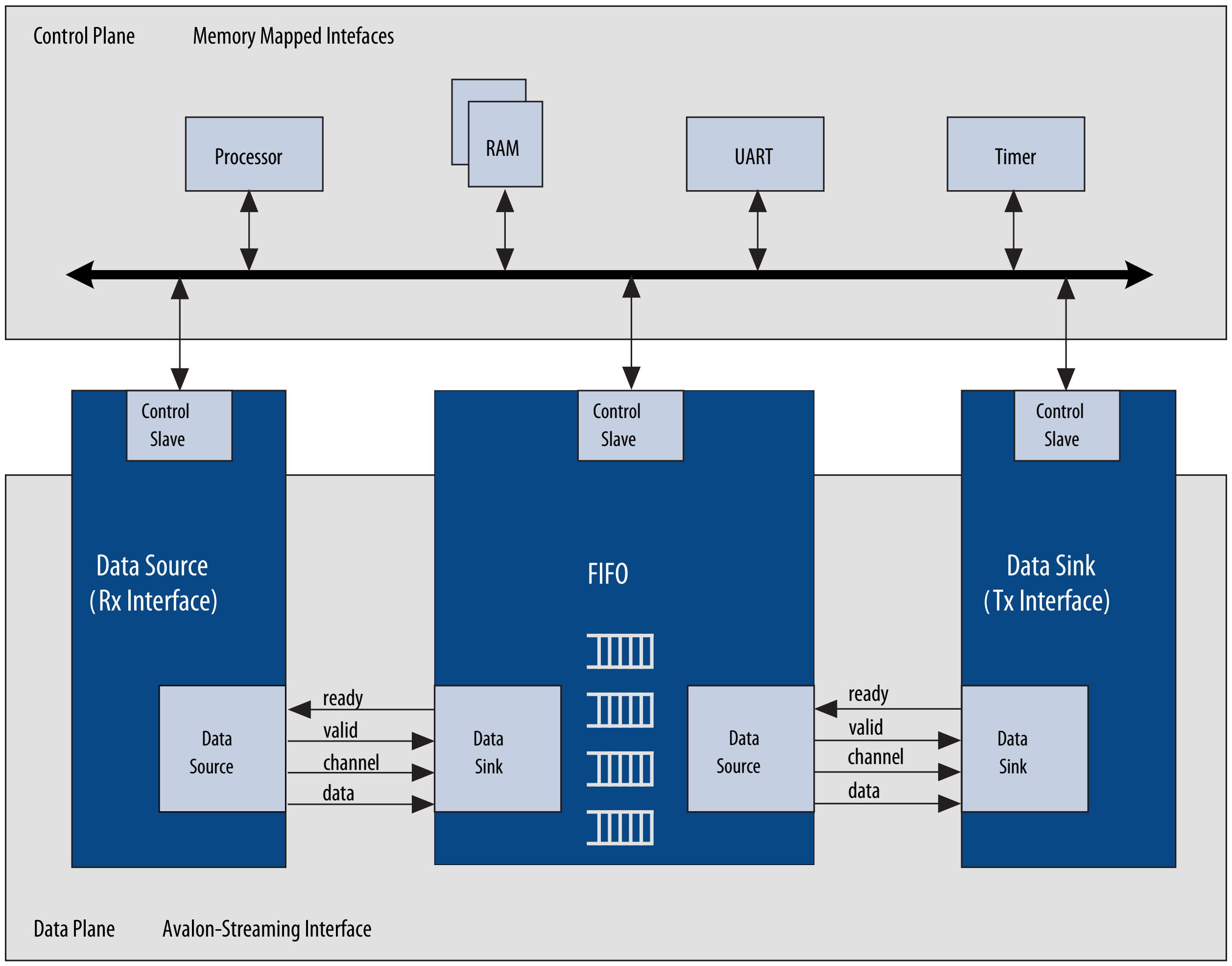


Figure 7-24: Avalon-ST Connection Between the Source and Sink

This source-sink pair includes only the `data` signal. The sink must be able to receive data as soon as the source interface comes out of reset.

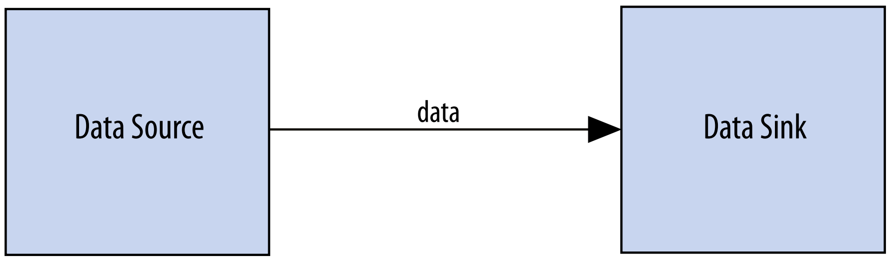
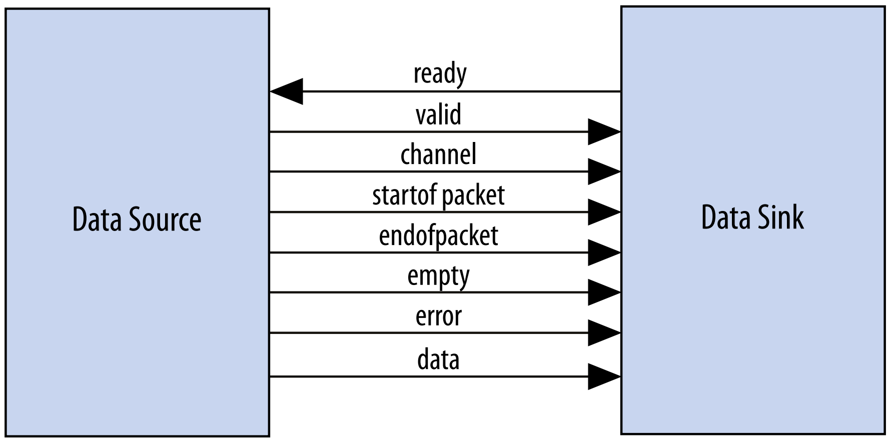


Figure 7-25: Signals Indicating the Start and End of Packets, Channel Numbers, Error Conditions, and Backpressure

All data transfers using Avalon-ST interconnect occur synchronously on the rising edge of the associated clock interface. Throughput and frequency of a system depends on the components and how they are connected.



The IP Catalog includes a number of Avalon-ST components that you can use to create datapaths, including datapaths whose input and output streams have different properties. Generated systems that include memory-mapped master and slave components may also use these Avalon-ST components because Qsys generation creates interconnect with a structure similar to a network topology, as described in *Qsys Transformations*. The following sections introduce the Avalon-ST components.

Related Information

- **Avalon-ST Adapters** on page 7-32
- **Qsys Transformations** on page 7-7
- **Avalon Interface Specification**

Avalon-ST Adapters

Qsys automatically adds Avalon-ST adapters between two components during system generation when it detects mismatched interfaces. If you connect mismatched Avalon-ST sources and sinks, for example, a

32-bit source and an 8-bit sink, Qsys inserts the appropriate adapter type to connect the mismatched interfaces.

After generation, you can view the inserted adapters with the **Show System With Qsys Interconnect** command in the System menu. For each mismatched source-sink pair, Qsys inserts an Avalon-ST Adapter. The adapter instantiates the necessary adaptation logic as sub-components. You can review the logic for each adapter instantiation in the Hierarchy view by expanding each adapter's source and sink interface and comparing the relevant ports. For example, to determine why a channel adapter is inserted, expand the channel adapter's sink and source interfaces and review the channel port properties for each interface.

You can turn off the auto-inserted adapters feature by adding the `qsys_enable_avalon_streaming_transform=off` command to the **quartus.ini** file. When you turn off the auto-inserted adapters feature, if mismatched interfaces are detected during system generation, Qsys does not insert adapters and reports the mismatched interface with validation error message.

Note: The auto-inserted adapters feature does not work for video IP core connections.

Avalon-ST Adapter

The Avalon-ST adapter combines the logic of the channel, error, data format, and timing adapters. The Avalon-ST adapter provides adaptations between interfaces that have mismatched Avalon-ST endpoints. Based on the source and sink interface parameterizations for the Avalon-ST adapter, Qsys instantiates the necessary adapter logic (channel, error, data format, or timing) as hierarchal sub-components.

Avalon-ST Adapter Parameters Common to Source and Sink Interfaces

Table 7-7: Avalon-ST Adapter Parameters Common to Source and Sink Interfaces

Parameter Name	Description
Symbol Width	Width of a single symbol in bits.
Use Packet	Indicates whether the source and sink interfaces connected to the adapter's source and sink interfaces include the `startofpacket` and `endofpacket` signals, and the optional `empty` signal.

Avalon-ST Adapter Upstream Source Interface Parameters

Table 7-8: Avalon-ST Adapter Upstream Source Interface Parameters

Parameter Name	Description
Source Data Width	Controls the data width of the source interface `data` port.
Source Top Channel	Maximum number of output channels allowed.
Source Channel Port Width	Sets the bit width of the source interface `channel` port. If set to 0, there is no `channel` port on the sink interface.
Source Error Port Width	Sets the bit width of the source interface `error` port. If set to 0, there is no `error` port on the sink interface.
Source Error Descriptors	A list of strings that describe the error conditions for each bit of the source interface `error` signal.

Parameter Name	Description
Source Uses Empty Port	Indicates whether the source interface includes the `empty` port, and whether the sink interface should also include the `empty` port.
Source Empty Port Width	Indicates the bit width of the source interface `empty` port, and sets the bit width of the sink interface `empty` port.
Source Uses Valid Port	Indicates whether the source interface connected to the sink interface uses the `valid` port, and if set, configures the sink interface to use the `valid` port.
Source Uses Ready Port	Indicates whether the sink interface uses the `ready` port, and if set, configures the source interface to use the `ready` port.
Source Ready Latency	Specifies what ready latency to expect from the source interface connected to the adapter's sink interface.

Avalon-ST Adapter Downstream Sink Interface Parameters

Table 7-9: Avalon-ST Adapter Downstream Sink Interface Parameters

Parameter Name	Description
Sink Data Width	Indicates the bit width of the `data` port on the sink interface connected to the source interface.
Sink Top Channel	Maximum number of output channels allowed.
Sink Channel Port Width	Indicates the bit width of the `channel` port on the sink interface connected the source interface.
Sink Error Port Width	Indicates the bit width of the `error` port on the sink interface connected to the adapter's source interface. If set to zero, there is no error port on the source interface.
Sink Error Descriptors	A list of strings that describe the error conditions for each bit of the `error` port on the sink interface connected to the source interface.
Sink Uses Empty Port	Indicates whether the sink interface connected to the source interface uses the `empty` port, and whether the source interface should also use the `empty` port.
Sink Empty Port Width	Indicates the bit width of the `empty` port on the sink interface connected to the source interface, and configures a corresponding `empty` port on the source interface.
Sink Uses Valid Port	Indicates whether the sink interface connected to the source interface uses the `valid` port, and if set, configures the source interface to use the `valid` port.
Sink Uses Ready Port	Indicates whether the `ready` port on the sink interface is connected to the source interface , and if set, configures the sink interface to use the ready port.

Parameter Name	Description
Sink Ready Latency	Specifies what ready latency to expect from the source interface connected to the sink interface.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different channel signal widths.

Table 7-10: Channel Adapter Adaptations

Condition	Description of Adapter Logic
The source uses channels, but the sink does not.	Qsys gives a warning at generation time. The adapter provides a simulation error and signals an error for data for any channel from the source other than 0.
The sink has channel, but the source does not.	Qsys gives a warning at generation time, and the channel inputs to the sink are all tied to a logical 0.
The source and sink both support channels, and the source's maximum channel number is less than the sink's maximum channel number.	The source's channel is connected to the sink's channel unchanged. If the sink's channel signal has more bits, the higher bits are tied to a logical 0.
The source and sink both support channels, but the source's maximum channel number is greater than the sink's maximum channel number.	The source’s channel is connected to the sink’s channel unchanged. If the source’s channel signal has more bits, the higher bits are left unconnected. Qsys gives a warning that channel information may be lost. An adapter provides a simulation error message and an error indication if the value of channel from the source is greater than the sink's maximum number of channels. In addition, the `valid` signal to the sink is deasserted so that the sink never sees data for channels that are out of range.

Avalon-ST Channel Adapter Input Interface Parameters

Table 7-11: Avalon-ST Channel Adapter Input Interface Parameters

Parameter Name	Description
Channel Signal Width (bits)	Width of the input channel signal in bits
Max Channel	Maximum number of input channels allowed.

Avalon-ST Channel Adapter Output Interface Parameters

Table 7-12: Avalon-ST Channel Adapter Output Interface Parameters

Parameter Name	Description
Channel Signal Width (bits)	Width of the output channel signal in bits.
Max Channel	Maximum number of output channels allowed.

Avalon-ST Channel Adapter Common to Input and Output Interface Parameters

Table 7-13: Avalon-ST Channel Adapter Common to Input and Output Interface Parameters

Parameter Name	Description
Data Bits Per Symbol	Number of bits for each symbol in a transfer.
Include Packet Support	When the Avalon-ST Channel adapter supports packets, the `startofpacket`, `endofpacket`, and optional `empty` signals are included on its sink and source interfaces.
Include Empty Signal	Indicates whether an `empty` signal is required.
Data Symbols Per Beat	Number of symbols per transfer.
Support Backpreasure with the ready signal	Indicates whether a `ready` signal is required.
Ready Latency	Specifies the ready latency to expect from the sink connected to the module's source interface.
Error Signal Width (bits)	Bit width of the `error` signal.
Error Signal Description	A list of strings that describes what each bit of the `error` signal represents.

Data Format Adapter

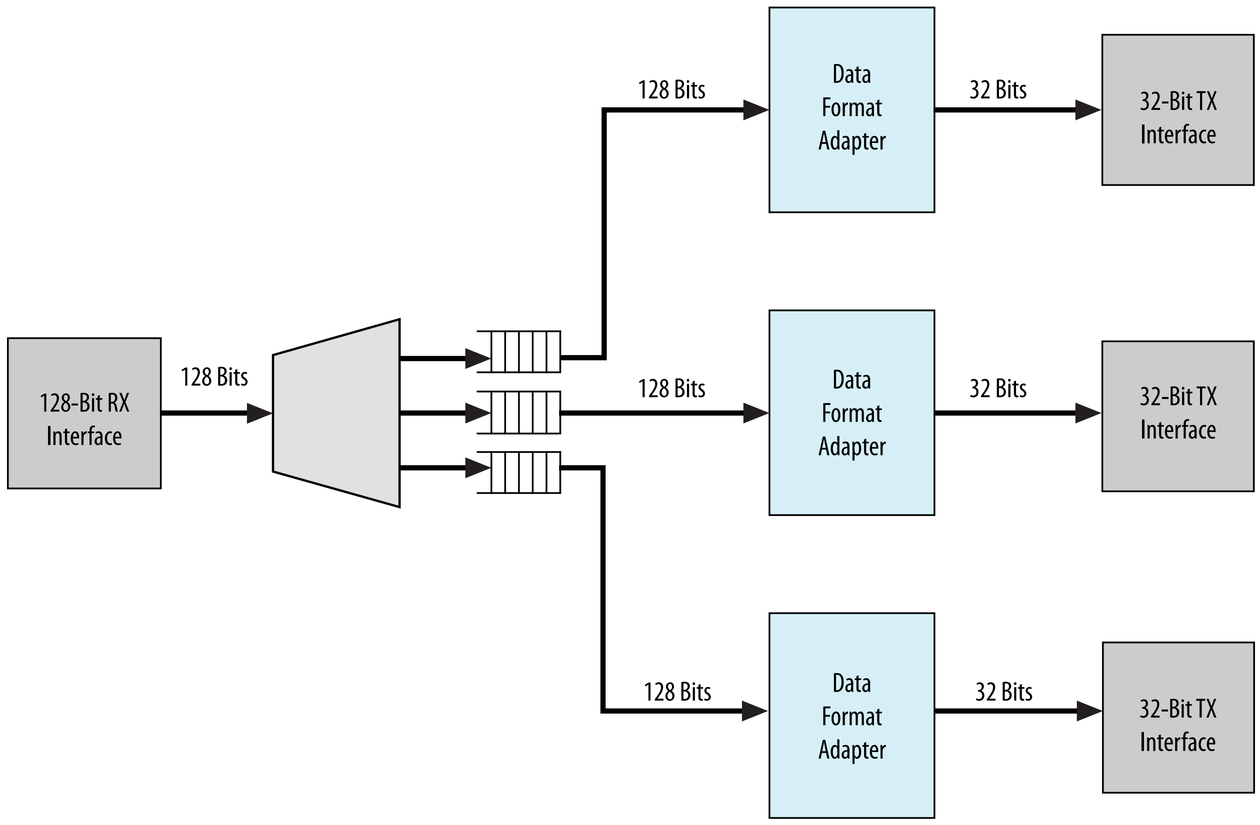
The data format adapter allows you to connect interfaces that have different values for the parameters defining the `data` signal, or interfaces where the source does not use the `empty` signal, but the sink does use the `empty` signal. One of the most common uses of this adapter is to convert data streams of different widths.

Table 7-14: Data Format Adapter Adaptations

Condition	Description of Adapter Logic
The source and sink's bits per symbol parameters are different.	The connection cannot be made.
The source and sink have a different number of symbols per beat.	The adapter converts the source's width to the sink's width. If the adaptation is from a wider to a narrower interface, a beat of data at the input corresponds to multiple beats of data at the output. If the input `error` signal is asserted for a single beat, it is asserted on output for multiple beats. If the adaptation is from a narrow to a wider interface, multiple input beats are required to fill a single output beat, and the output `error` is the logical OR of the input `error` signal.
The source uses the `empty` signal, but the sink does not use the `empty` signal.	Qsys cannot make the connection.

Figure 7-26: Avalon Streaming Interconnect with Data Format Adapter

In this example, the data format adapter allows a connection between a 128-bit output data stream and three 32-bit input data streams.



Avalon-ST Data Format Adapter Input Interface Parameters

Table 7-15: Avalon-ST Data Format Adapter Input Interface Parameters

Parameter Name	Description
Data Symbols Per Beat	Number of symbols per transfer.
Include Empty Signal	Indicates whether an `empty` signal is required.

Avalon-ST Data Format Adapter Output Interface Parameters

Table 7-16: Avalon-ST Data Format Adapter Output Interface Parameters

Parameter Name	Description
Data Symbols Per Beat	Number of symbols per transfer.
Include Empty Signals	Indicates whether an `empty` signal is required.

Avalon-ST Data Format Adapter Common to Input and Output Interface Parameters

Table 7-17: Avalon-ST Data Format Adapter Common to Input and Output Interface Parameters

Parameter Name	Description
Data Bits Per Symbol	Number of bits for each symbol in a transfer.
Include Packet Support	When the Avalon-ST Data Format adapter supports packets, Qsys uses `startofpacket`, `endofpacket`, and `empty` signals.
Channel Signal Width (bits)	Width of the output channel signal in bits.
Max Channel	Maximum number of channels allowed.
Read Latency	Specifies the ready latency to expect from the sink connected to the module's source interface.
Error Signal Width (bits)	Width of the `error` signal output in bits.
Error Signal Description	A list of strings that describes what each bit of the `error` signal represents.

Error Adapter

The error adapter ensures that per-bit-error information provided by the source interface is correctly connected to the sink interface's input error signal. Error conditions that both the source and sink are able to process are connected. If the source has an `error` signal representing an error condition that is not supported by the sink, the signal is left unconnected; the adapter provides a simulation error message and an error indication if the error is asserted. If the sink has an error condition that is not supported by the source, the sink's input error bit corresponding to that condition is set to 0.

Note: The output interface error signal descriptor accepts an error set with an `other` descriptor. Qsys assigns the bit-wise `ORing` of all input error bits that are unmatched, to the output interface error bits set with the `other` descriptor.

Avalon-ST Error Adapter Input Interface Parameters

Table 7-18: Avalon-ST Error Adapter Input Interface Parameters

Parameter Name	Description
Error Signal Width (bits)	The width of the `error` signal. Valid values are 0–256 bits. Type `0` if the `error` signal is not used.
Error Signal Description	The description for each of the error bits. If scripting, separate the description fields by commas. For a successful connection, the description strings of the error bits in the source and sink must match and are case sensitive.

Avalon-ST Error Adapter Output Interface Parameters

Table 7-19: Avalon-ST Error Adapter Output Interface Parameters

Parameter Name	Description
Error Signal Width (bits)	The width of the `error` signal. Valid values are 0–256 bits. Type `0` if you do not need to send error values.
Error Signal Description	The description for each of the error bits. Separate the description fields by commas. For successful connection, the description of the error bits in the source and sink must match, and are case sensitive.

Avalon-ST Error Adapter Common to Input and Output Interface Parameters

Table 7-20: Avalon-ST Error Adapter Common to Input and Output Interface Parameters

Parameter Name	Description
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the ready signal is asserted and when valid data is driven.
Channel Signal Width (bits)	The width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	The maximum number of channels that the interface supports. Valid values are 0–255.
Data Bits Per Symbol	Number of bits per symbol.
Data Symbols Per Beat	Number of symbols per active transfer.
Include Packet Support	Turn on this option if the connected interfaces support a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.

Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO buffer between the source and sink to buffer data or pipeline stages to delay the back pressure signals. You can also use the timing adapter to connect interfaces that support the `ready` signal, and those that do not. The timing adapter treats all signals other than the `ready` and `valid` signals as payload, and simply drives them from the source to the sink.

Table 7-21: Timing Adapter Adaptations

Condition	Adaptation
The source has `ready`, but the sink does not.	In this case, the source can respond to `backpressure`, but the sink never needs to apply it. The `ready` input to the source interface is connected directly to logical 1.
The source does not have `ready`, but the sink does.	The sink may apply `backpressure`, but the source is unable to respond to it. There is no logic that the adapter can insert that prevents data loss when the source asserts `valid` but the sink is not ready. The adapter provides simulation time error messages if data is lost. The user is presented with a warning, and the connection is allowed.
The source and sink both support backpressure, but the sink's ready latency is greater than the source's.	The source responds to `ready` assertion or deassertion faster than the sink requires it. A number of pipeline stages equal to the difference in ready latency are inserted in the `ready` path from the sink back to the source, causing the source and the sink to see the same cycles as `ready` cycles.
The source and sink both support backpressure, but the sink's ready latency is less than the source's.	The source cannot respond to `ready` assertion or deassertion in time to satisfy the sink. A FIFO whose depth is equal to the difference in ready latency is inserted to compensate for the source's inability to respond in time.

Avalon-ST Timing Adapter Input Interface Parameters

Table 7-22: Avalon-ST Timing Adapter Input Interface Parameters

Parameter Name	Description
Support Backpreasure with the ready signal	Indicates whether a `ready` signal is required.
Read Latency	Specifies the ready latency to expect from the sink connected to the module's source interface.

Parameter Name	Description
Include Valid Signal	Indicates whether the sink interface requires a valid signal.

Avalon-ST Timing Adapter Output Interface Parameters

Table 7-23: Avalon-ST Timing Adapter Output Interface Parameters

Parameter Name	Description
Support Backpreasure with the ready signal	Indicates whether a `ready` signal is required.
Read Latency	Specifies the ready latency to expect from the sink connected to the module's source interface.
Include Valid Signal	Indicates whether the sink interface requires a valid signal.

Avalon-ST Timing Adapter Common to Input and Output Interface Parameters

Table 7-24: Avalon-ST Timing Adapter Common to Input and Output Interface Parameters

Parameter Name	Description
Data Bits Per Symbol	Number of bits for each symbol in a transfer.
Include Packet Support	Turn this option on if the connected interfaces support a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.
Data Symbols Per Beat	Number of symbols per active transfer.
Channel Signal Width (bits)	Width of the output channel signal in bits.
Max Channel	Maximum number of output channels allowed.
Error Signal Width (bits)	Width of the output `error` signal in bits.
Error Signal Description	A list of strings that describes errors.

Interrupt Interfaces

Using individual requests, the interrupt logic can process up to 32 IRQ inputs connected to each interrupt receiver. With this logic, the interrupt sender connected to interrupt `receiver_0` is the highest priority with sequential receivers being successively lower priority. You can redefine the priority of interrupt senders by instantiating the IRQ mapper component. For more information refer to *IRQ Mapper*.

You can define the interrupt sender interface as asynchronous with no associated clock or reset interfaces. You can also define the interrupt receiver interface as asynchronous with no associated clock or reset interfaces. As a result, the receiver does its own synchronization internally. Qsys does not insert interrupt synchronizers for such receivers.

For clock crossing adaption on interrupts, Qsys inserts a synchronizer, which is clocked with the interrupt end point interface clock when the corresponding starting point interrupt interface has no clock or a different clock (than the end point). Qsys inserts the adapter if there is any kind of mismatch between the start and end points. Qsys does not insert the adapter if the interrupt receiver does not have an associated clock.

Related Information

IRQ Mapper on page 7-45

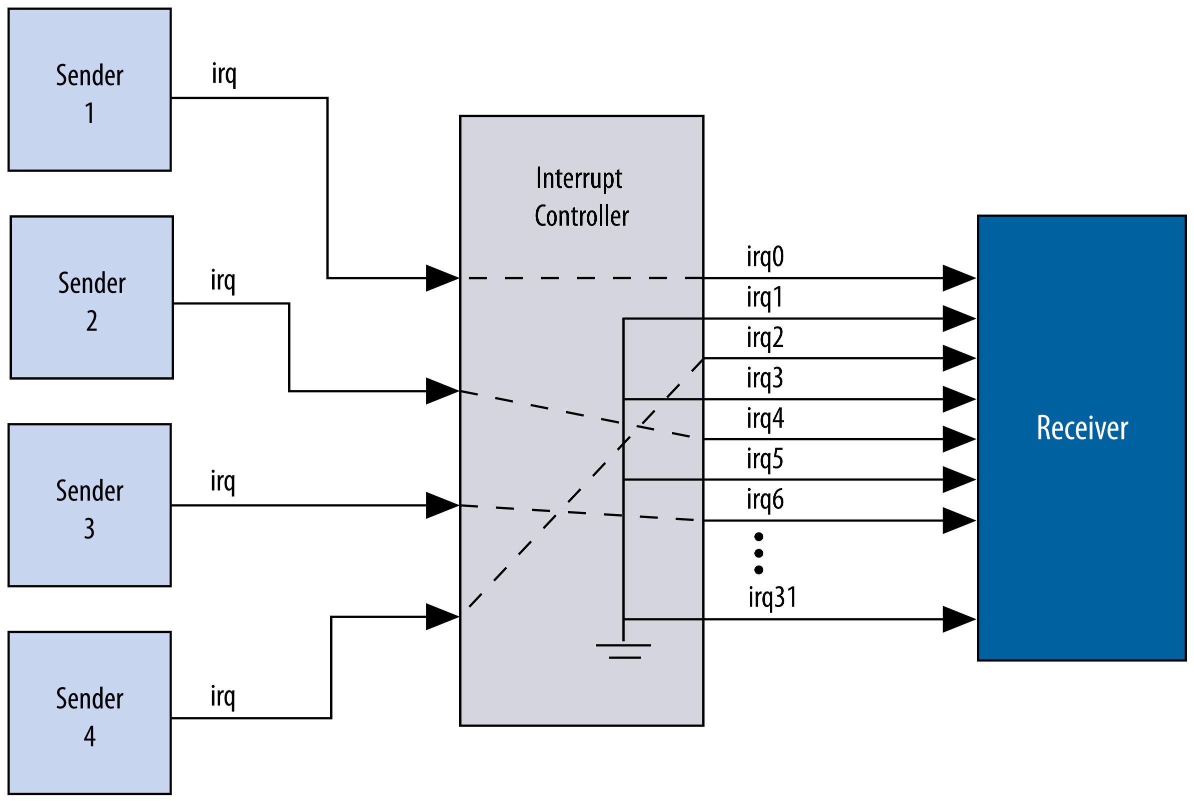
Individual Requests IRQ Scheme

In the individual requests IRQ scheme, Qsys interconnect passes IRQs directly from the sender to the receiver, without making assumptions about IRQ priority. In the event that multiple senders assert their

IRQs simultaneously, the receiver logic determines which IRQ has highest priority, and then responds appropriately.

Figure 7-27: Interrupt Controller Mapping IRQs

Using individual requests, the interrupt controller can process up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31:0]` to the receiver, and maps slave IRQ signals to the bits of `irq[31:0]`. Any unassigned bits of `irq[31:0]` are disabled.



Assigning IRQs in Qsys

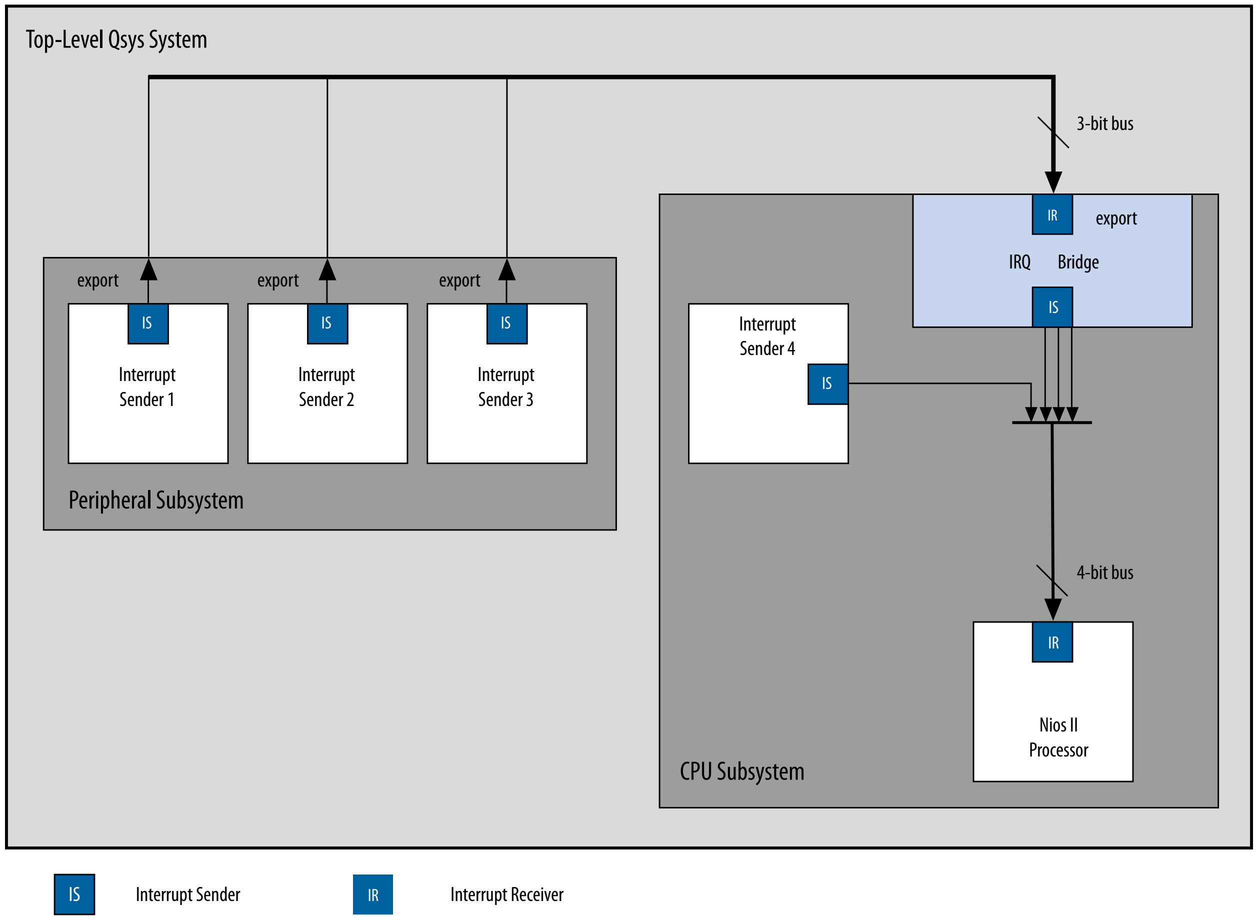
You assign IRQ connections on the **System Contents** tab of Qsys. After adding all components to the system, you connect interrupt senders and receivers. You can use the **IRQ** column to specify an IRQ number with respect to each receiver, or to specify a receiver's IRQ as unconnected. Qsys uses the following three components to implement interrupt handling: IRQ Bridge, IRQ Mapper, and IRQ Clock Crosser.

IRQ Bridge

The IRQ Bridge allows you to route interrupt wires between Qsys subsystems.

Figure 7-28: Qsys IRQ Bridge Application

The peripheral subsystem example below has three interrupt senders that are exported to the to- level of the subsystem. The interrupts are then routed to the CPU subsystem using the IRQ bridge.



Note: Nios II BSP tools support the IRQ Bridge. Interrupts connected via an IRQ Bridge appear in the generated **system.h** file. You can use the following properties with the IRQ Bridge, which do not effect Qsys interconnect generation. Qsys uses these properties to generate the correct IRQ information for downstream tools:

- **`set_interface_property` <sender port> `bridgesToReceiver` <receiver port>**—The *<sender port>* of the IP generates a signal that is received on the IP's *<receiver port>*. Sender ports are single bits. Receivers ports can be multiple bits. Qsys requires the `bridgedReceiverOffset` property to identify the *<receiver port>* bit that the *<sender port>* sends.
- **`set_interface_property` <sender port> `bridgedReceiverOffset` <port number>**—Indicates the <port number> of the receiver port that the *<sender port>* sends.

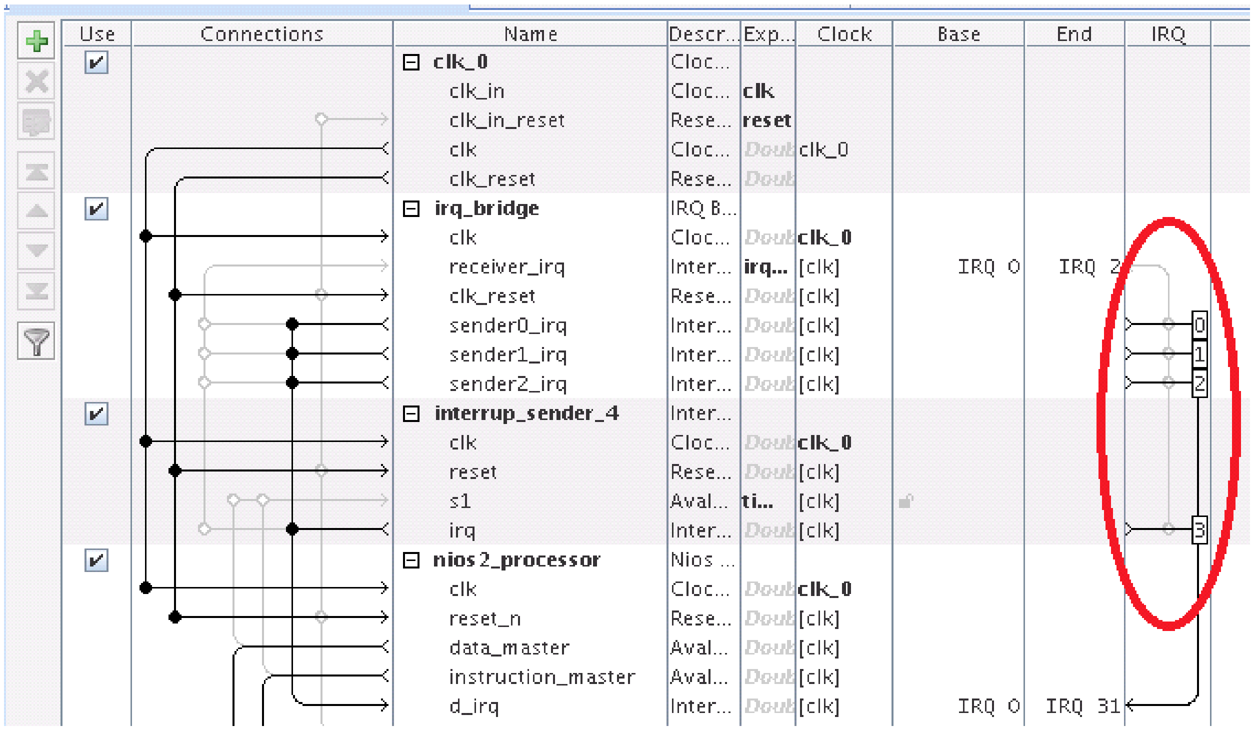
IRQ Mapper

Qsys inserts the IRQ Mapper automatically during generation. The IRQ Mapper converts individual interrupt wires to a bus, and then maps the appropriate IRQ priority number onto the bus.

By default, the interrupt sender connected to the `receiver0` interface of the IRQ mapper is the highest priority, and sequential receivers are successively lower priority. You can modify the interrupt priority of each IRQ wire by modifying the IRQ priority number in Qsys under the **IRQ** column. The modified priority is reflected in the **IRQ_MAP** parameter for the auto-inserted IRQ Mapper.

Figure 7-29: IRQ Column in Qsys

Circled in the **IRQ** column are the default interrupt priorities allocated for the CPU subsystem.



Use	Connections	Name	Descr...	Exp...	Clock	Base	End	IRQ
✓		⊟ **clk_0**	Cloc...					
		clk_in	Cloc...	**clk**				
		clk_in_reset	Rese...	**reset**				
		clk	Cloc...	Doub	clk_0			
		clk_reset	Rese...	Doub				
✓		⊟ **irq_bridge**	IRQ B...					
		clk	Cloc...	Doub	**clk_0**			
		receiver_irq	Inter...	**irq...**	[clk]	IRQ 0	IRQ 2	
		clk_reset	Rese...	Doub	[clk]			
		sender0_irq	Inter...	Doub	[clk]			0
		sender1_irq	Inter...	Doub	[clk]			1
		sender2_irq	Inter...	Doub	[clk]			2
✓		⊟ **interrupt_sender_4**	Inter...					
		clk	Cloc...	Doub	**clk_0**			
		reset	Rese...	Doub	[clk]			
		s1	Aval...	**ti...**	[clk]			
		irq	Inter...	Doub	[clk]			3
✓		⊟ **nios2_processor**	Nios ...					
		clk	Cloc...	Doub	**clk_0**			
		reset_n	Rese...	Doub	[clk]			
		data_master	Aval...	Doub	[clk]			
		instruction_master	Aval...	Doub	[clk]			
		d_irq	Inter...	Doub	[clk]	IRQ 0	IRQ 31	

Related Information

IRQ Bridge on page 7-44

IRQ Clock Crosser

The IRQ Clock Crosser synchronizes interrupt senders and receivers that are in different clock domains. To use this component, connect the clocks for both the interrupt sender and receiver, and for both the interrupt sender and receiver interfaces. Qsys automatically inserts this component when it is required.

Clock Interfaces

Clock interfaces define the clocks used by a component. Components can have clock inputs, clock outputs, or both. To update the clock frequency of the component, use the **Parameters** tab for the clock source.

The **Clock Source** parameters allows you to set the following options:

- **Clock frequency**—The frequency of the output clock from this clock source.
- **Clock frequency is known**— When turned on, the clock frequency is known. When turned off, the frequency is set from outside the system.

  Note: If turned off, system generation may fail because the components do not receive the necessary clock information. For best results, turn this option on before system generation.
- **Reset synchronous edges**
  - **None**—The reset is asserted and deasserted asynchronously. You can use this setting if you have internal synchronization circuitry that matches the reset required for the IP in the system.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously and asserted asynchronously.

For more information about synchronous design practices, refer to *Recommended Design Practices*

Related Information

- Recommended Design Practices on page 11-1

(High Speed Serial Interface) HSSI Clock Interfaces

You can use HSSI Serial Clock and HSSI Bonded Clock interfaces in Qsys to enable high speed serial connectivity between clocks that are used by certain IP protocols.

HSSI Serial Clock Interface

You can connect the HSSI Serial Clock interface with only similar type of interfaces, for example, you can connect a HSSI Serial Clock Source interface to a HSSI Serial Clock Sink interface.

HSSI Serial Clock Source

The HSSI Serial Clock interface includes a source in the **Start** direction.

You can instantiate the HSSI Serial Clock Source interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_serial_clock start
```

You can connect the HSSI Serial Clock Source to multiple HSSI Serial Clock Sinks because the HSSI Serial Clock Source supports multiple fan-outs. This Interface has a single **clk** port role limited to a 1 bit width, and a **clockRate** parameter, which is the frequency of the clock driven by the HSSI Serial Clock Source interface.

An unconnected and unexported HSSI Serial Source is valid and does not generate error messages.

Table 7-25: HSSI Serial Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 bit	A single bit wide port role, which provides synchronization for internal logic.

Table 7-26: HSSI Serial Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven byte HSSI Serial Clock Source interface.

HSSI Serial Clock Sink

The HSSI Serial Clock interface includes a sink in the **End** direction.

You can instantiate the HSSI Serial Clock Sink interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_serial_clock end
```

You can connect the HSSI Serial Clock Sink interface to a single HSSI Serial Clock Source interface; you cannot connect it to multiple sources. This Interface has a single **clk** port role limited to a 1 bit width, and a **clockRate** parameter, which is the frequency of the clock driven by the HSSI Serial Clock Source interface.

An unconnected and unexported HSSI Serial Sink is invalid and generates error messages.

Table 7-27: HSSI Serial Clock Sink Port Roles

Name	Direction	Width	Description
clk	Output	1	A single bit wide port role, which provides synchronization for internal logic

Table 7-28: HSSI Serial Clock Sink Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven by the HSSI Serial Clock Source interface. When you specify a **clockRate** greater than 0, then this interface can be driven only at that rate.

HSSI Serial Clock Connection

The HSSI Serial Clock Connection defines a connection between a HSSI Serial Clock Source connection point, and a HSSI Serial Clock Sink connection point.

A valid HSSI Serial Clock Connection exists when all of the following criteria are satisfied. If the following criteria are not satisfied, Qsys generates error messages and the connection is prohibited.

- The starting connection point is an HSSI Serial Clock Source with a single port role **clk** and maximum 1 bit in width. The direction of the starting port is **Output**.
- The ending connection point is an HSSI Serial Clock Sink with a single port role **clk**, and maximum 1 bit in width. The direction of the ending port is **Input**.
- If the parameter, **clockRate** of the HSSI Serial Clock Sink is greater than 0, the connection is only valid if the **clockRate** of the HSSI Serial Clock Source is the same as the **clockRate** of the HSSI Serial Clock Sink.

HSSI Serial Clock Example

Example 7-1: HSSI Serial Clock Interface Example

You can make connections to declare the HSSI Serial Clock interfaces in the **_hw.tcl**.

```
package require -exact qsys 14.0

set_module_property name hssi_serial_component
set_module_property ELABORATION_CALLBACK elaborate

add_fileset QUARTUS_SYNTH QUARTUS_SYNTH generate
add_fileset SIM_VERILOG SIM_VERILOG generate
add_fileset SIM_VHDL SIM_VHDL generate

set_fileset_property QUARTUS_SYNTH TOP_LEVEL \
"hssi_serial_component"

set_fileset_property SIM_VERILOG TOP_LEVEL "hssi_serial_component"
set_fileset_property SIM_VHDL TOP_LEVEL "hssi_serial_component"

proc elaborate {} {
    # declaring HSSI Serial Clock Source
    add_interface my_clock_start hssi_serial_clock start
    set_interface_property my_clock_start  ENABLED true

    add_interface_port my_clock_start  hssi_serial_clock_port_out \
 clk Output 1

    # declaring HSSI Serial Clock Sink
    add_interface my_clock_end hssi_serial_clock end
    set_interface_property my_clock_end  ENABLED true

    add_interface_port my_clock_end  hssi_serial_clock_port_in clk \
 Input 1
}

proc generate { output_name } {

    add_fileset_file hssi_serial_component.v VERILOG PATH \
 "hssi_serial_component.v"
}
```

Example 7-2: HSSI Serial Clock Instantiated in a Composed Component

If you use the components in a hierarchy, for example, instantiated in a composed component, you can declare the connections as illustrated in this example.

```
add_instance myinst1 hssi_serial_component
add_instance myinst2 hssi_serial_component
# add connection from source of myinst1 to sink of myinst2

add_connection myinst1.my_clock_start myinst2.my_clock_end \
hssi_serial_clock

# adding connection from source of myinst2 to sink of myinst1

add_connection myinst2.my_clock_start myinst2.my_clock_end \
hssi_serial_clock
```

HSSI Bonded Clock Interface

You can connect the HSSI Bonded Clock interface with only similar type of Interfaces, for example, you can connect a HSSI Bonded Clock Source interface to a HSSI Bonded Clock Sink interface.

HSSI Bonded Clock Source

The HSSI Bonded Clock interface includes a source in the **Start** direction.

You can instantiate the HSSI Bonded Clock Source interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_bonded_clock start
```

You can connect the HSSI Bonded Clock Source to multiple HSSI Bonded Clock Sinks because the HSSI Serial Clock Source supports multiple fanouts. This Interface has a single **clk** port role limited to a width range of 1 to 1024 bits. The HSSI Bonded Clock Source interface has two parameters: **clockRate** and **serialzationFactor**. **clockRate** is the frequency of the clock driven by the HSSI Bonded Clock Source interface, and the **serializationFactor** is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the required frequency and phases of the individual clocks within the HSSI Bonded Clock interface

An unconnected and unexported HSSI Bonded Source is valid and does not generate error messages.

Table 7-29: HSSI Bonded Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 to 24 bits	A multiple bit wide port role which provides synchronization for internal logic.

Table 7-30: HSSI Bonded Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven byte HSSI Serial Clock Source interface.
serialization	long	0	No	The serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the necessary frequency and phases of the individual clocks within the HSSI Bonded Clock interface.

HSSI Bonded Clock Sink

The HSSI Bonded Clock interface includes a sink in the **End** direction.

You can instantiate the HSSI Bonded Clock Sink interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_bonded_clock end
```

You can connect the HSSI Bonded Clock Sink interface to a single HSSI Bonded Clock Source interface; you cannot connect it to multiple sources. This Interface has a single **clk** port role limited to a width range of 1 to 1024 bits. The HSSI Bonded Clock Source interface has two parameters: **clockRate** and **serialza-**

tionFactor. **clockRate** is the frequency of the clock driven by the HSSI Bonded Clock Source interface, and the serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the required frequency and phases of the individual clocks within the HSSI Bonded Clock interface

An unconnected and unexported HSSI Bonded Sink is invalid and generates error messages.

Table 7-31: HSSI Bonded Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 to 24 bits	A multiple bit wide port role which provides synchronization for internal logic.

Table 7-32: HSSI Bonded Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven byte HSSI Serial Clock Source interface.
serialization	long	0	No	The serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the necessary frequency and phases of the individual clocks within the HSSI Bonded Clock interface.

HSSI Bonded Clock Connection

The HSSI Bonded Clock Connection defines a connection between a HSSI Bonded Clock Source connection point, and a HSSI Bonded Clock Sink connection point.

A valid HSSI Bonded Clock Connection exists when all of the following criteria are satisfied. If the following criteria are not satisfied, Qsys generates error messages and the connection is prohibited.

- The starting connection point is an HSSI Bonded Clock Source with a single port role **clk** with a width range of 1 to 24 bits. The direction of the starting port is **Output**.
- The ending connection point is an HSSI Bonded Clock Sink with a single port role **clk** with a width range of 1 to 24 bits. The direction of the ending port is **Input**.
- The width of the starting connection point **clk** must be the same as the width of the ending connection point.
- If the parameter, **clockRate** of the HSSI Bonded Clock Sink greater than 0, then the connection is only valid if the **clockRate** of the HSSI Bonded Clock Source is same as the **clockRate** of the HSSI Bonded Clock Sink.
- If the parameter, **serializationFactor** of the HSSI Bonded Clock Sink is greater than 0, Qsys generates a warning if the **serializationFactor** of HSSI Bonded Clock Source is not same as the **serialization-Factor** of the HSSI Bonded Clock Sink.

HSSI Bonded Clock Example

Example 7-3: HSSI Bonded Clock Interface Example

You can make connections to declare the HSSI Bonded Clock interfaces in the **_hw.tcl** file.

```
package require -exact qsys 14.0

set_module_property name hssi_bonded_component
set_module_property ELABORATION_CALLBACK elaborate

add_fileset synthesis QUARTUS_SYNTH generate
add_fileset verilog_simulation SIM_VERILOG generate

set_fileset_property synthesis TOP_LEVEL "hssi_bonded_component"

set_fileset_property verilog_simulation TOP_LEVEL \
"hssi_bonded_component"

proc elaborate {} {
    add_interface my_clock_start hssi_bonded_clock start
    set_interface_property my_clock_start  ENABLED true

    add_interface_port my_clock_start  hssi_bonded_clock_port_out \
 clk Output 1024

    add_interface my_clock_end hssi_bonded_clock end
    set_interface_property my_clock_end  ENABLED true

    add_interface_port my_clock_end  hssi_bonded_clock_port_in \
 clk Input 1024
}

proc generate { output_name } {
    add_fileset_file hssi_bonded_component.v VERILOG PATH \
 "hssi_bonded_component.v"}
```

If you use the components in a hierarchy, for example, instantiated in a composed component, you can declare the connections as illustrated in this example.

Example 7-4: HSII Bonded Clock Instantiated in a Composed Component

```
add_instance myinst1 hssi_bonded_component
add_instance myinst2 hssi_bonded_component
# add connection from source of myinst1 to sink of myinst2

add_connection myinst1.my_clock_start myinst2.my_clock_end \
hssi_bonded_clock

# adding connection from source of myinst2 to sink of myinst1

add_connection myinst2.my_clock_start myinst2.my_clock_end \
hssi_bonded_clock
```

Reset Interfaces

Reset interfaces provide both soft and hard reset functionality. Soft reset logic typically re-initializes registers and memories without powering down the device. Hard reset logic initializes the device after power-on. You can define separate reset sources for each clock domain, a single reset source for all clocks, or any combination in between.

You can choose to create a single global reset domain by selecting **Create Global Reset Network** on the System menu. If your design requires more than one reset domain, you can implement your own reset logic and connectivity. The IP Catalog includes a reset controller, reset sequencer, and a reset bridge to implement the reset functionality. You can also design your own reset logic.

Note: If you design your own reset circuitry, you must carefully consider situations which may result in system lockup. For example, if an Avalon-MM slave is reset in the middle of a transaction, the Avalon-MM master may lockup.

Single Global Reset Signal Implemented by Qsys

If you select **Create Global Reset Network** on the System menu, the Qsys interconnect creates a global reset bus. All of the reset requests are ORed together, synchronized to each clock domain, and fed to the reset inputs. The duration of the reset signal is at least one clock period.

The Qsys interconnect inserts the system-wide reset under the following conditions:

- The global reset input to the Qsys system is asserted.
- Any component asserts its `resetrequest` signal.

Reset Controller

Qsys automatically inserts a reset controller block if the input reset source does not have a reset request, but the connected reset sink requires a reset request.

The Reset Controller has the following parameters that you can specify to customize its behavior:

- **Number of inputs**— Indicates the number of individual reset interfaces the controller ORs to create a signal reset output.
- **Output reset synchronous edges**—Specifies the level of synchronization. You can select one the following options:
  - **None**—The reset is asserted and deasserted asynchronously. You can use this setting if you have designed internal synchronization circuitry that matches the reset style required for the IP in the system.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously and asserted asynchronously.
- **Synchronization depth**—Specifies the number of register stages the synchronizer uses to eliminate the propagation of metastable events.
- **Reset request**—Enables reset request generation, which is an early signal that is asserted before reset assertion. The reset request is used by blocks that require protection from asynchronous inputs, for example, M20K blocks.

Qsys automatically inserts reset synchronizers under the following conditions:

- More than one reset source is connected to a reset sink
- There is a mismatch between the reset source's synchronous edges and the reset sinks' synchronous edges

Reset Bridge

The Reset Bridge allows you to use a reset signal in two or more subsystems of your Qsys system. You can connect one reset source to local components, and export one or more to other subsystems, as required.

The Reset Bridge parameters are used to describe the incoming reset and include the following options:

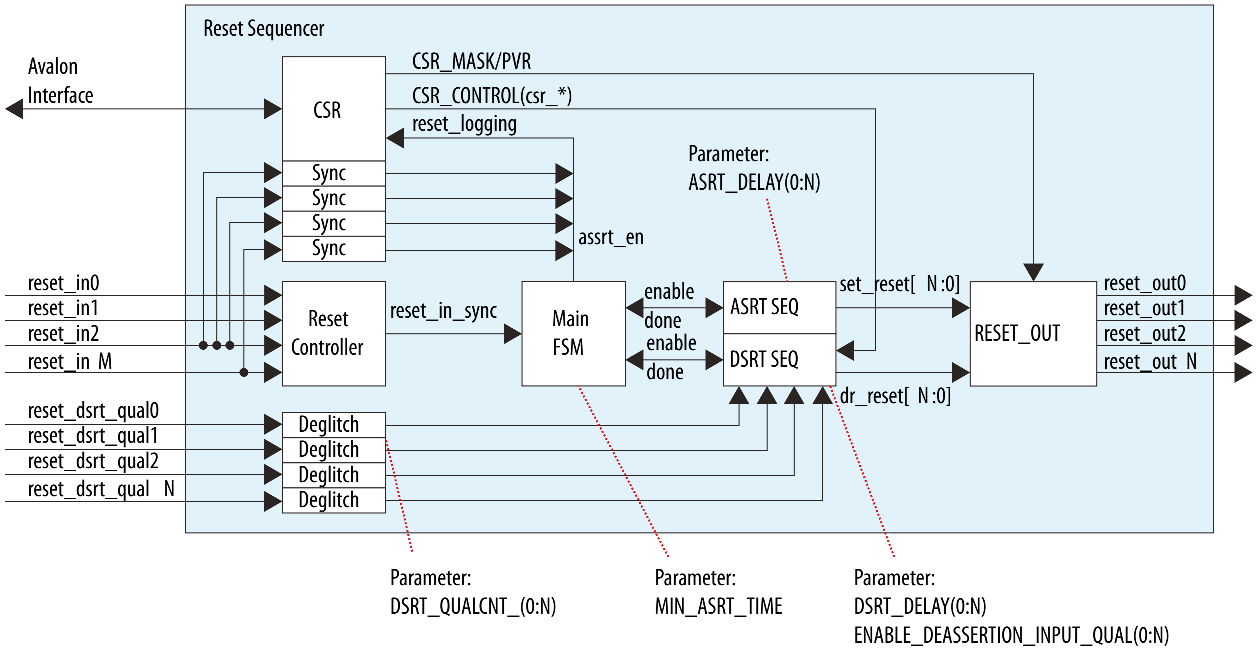
- **Active low reset**—When turned on, reset is asserted low.
- **Synchronous edges**—Specifies the level of synchronization and includes the following options:
  - **None**—The reset is asserted and deasserted asynchronously. Use this setting if you have internal synchronization circuitry.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously, and asserted asynchronously.
- **Number of reset outputs**—The number of reset interfaces that are exported.

Note: Qsys supports multiple reset sink connections to a single reset source interface. However, there are situations in composed systems where an internally generated reset must be exported from the composed system in addition to being used to connect internal components. In this situation, you must declare one reset output interface as an export, and use another reset output to connect internal components.

Reset Sequencer

The Reset Sequencer allows you to control the assertion and de-assertion sequence for Qsys system resets. The Parameter Editor displays the expected assertion and de-assertion sequences based on the current settings. You can connect multiple reset sources to the reset sequencer, and then connect the output of the reset sequencer to components in the system.

Figure 7-30: Elements and Flow of a Reset Sequencer



- Reset Controller —Reused reset controller block. It synchronizes the reset inputs into one and feed into the main FSM of the sequencer block.
- Sync —Synchronization block (double flip-flop).
- Deglitch —Deglitch block. This block waits for a signal to be at a level for X clocks before propagating the input to the output.
- CSR —This block contains the CSR Avalon interface and related CSR register and control block in the sequencer.
- Main FSM —Main sequencer. This block determines when assertion/deassertion and assertion hold timing occurs.
- [A/D]SRT SEQ —Generic sequencer block that sequences out assertion/deassertion of reset from 0:N. The block has multiple counters that saturate upon reaching count.
- RESET_OUT —Controls the end output via:
  - Set/clear from the ASRT_SEQ/DSRT_SEQ.
  - Masking/forcing from CSR controls.
  - Remap of numbering (parameterization).

Reset Sequencer Parameters

Table 7-33: Reset Sequencer Parameters

Parameter	Description
Number of reset outputs	Sets the number of output resets to be sequenced, which is the number of output reset signals defined in the component with a range of 2 to 10.
Number of reset inputs	Sets the number of input reset signals to be sequenced, which is the number of input reset signals defined in the component with a range of 1 to 10.
Minimum reset assertion time	Specifies the minimum assertion cycles between the assertion of the last sequenced reset, and the de-assertion of the first sequenced reset. The range is 0 to 1023.

Parameter	Description
Enable Reset Sequencer CSR	Enables CSR functionality of the Reset Sequencer through an Avalon interface.
reset_out#	Lists the reset output signals. Set the parameters in the other columns for each reset signal in the table.
ASRT Seq#	Determines the order of reset assertion. Enter the values 1, 2, 3, etc. to specify the required non-overlapping assertion order. This value determines the `ASRT_REMAP` value in the component HDL.
ASRT Cycle#	Number of cycles to wait before assertion of the reset. The value set here corresponds to the `ASRT_DELAY` value in the component HDL.The range is 0 to1023.
DSRT Seq#	Determines the reset order of reset de-assertion. Enter the values 1, 2, 3, etc .to specify the required non-overlapping de-assertion order. This value determines the `DSRT_REMAP` value in the component HDL.
DSRT Cycle#/Deglitch#	Number of cycles to wait before de-asserting or de-glitching the reset. If the **USE_DRST_QUAL** parameter is set to 0, specifies the number of cycles to wait before de-asserting the reset. If **USE_DSRT_QUAL** is set to1, specifies the number of cycles to deglitch the input `reset_dsrt_qual` signal. This value determines either the `DSRT_DELAY`, or the `DSRT_QUALCNT` value in the component HDL, depending on the **USE_DSRT_QUAL** parameter setting. The range is 0 to 1023.
USE_DSRT_QUAL	If you set **USE_DSRT_QUAL** to 1, the de-assertion sequence waits for an external input signal for sequence qualification instead of waiting for a fixed delay count. To use a fixed delay count for de-assertion, set this parameter to 0.

Reset Sequencer Timing Diagrams

Figure 7-31: Basic Sequencing

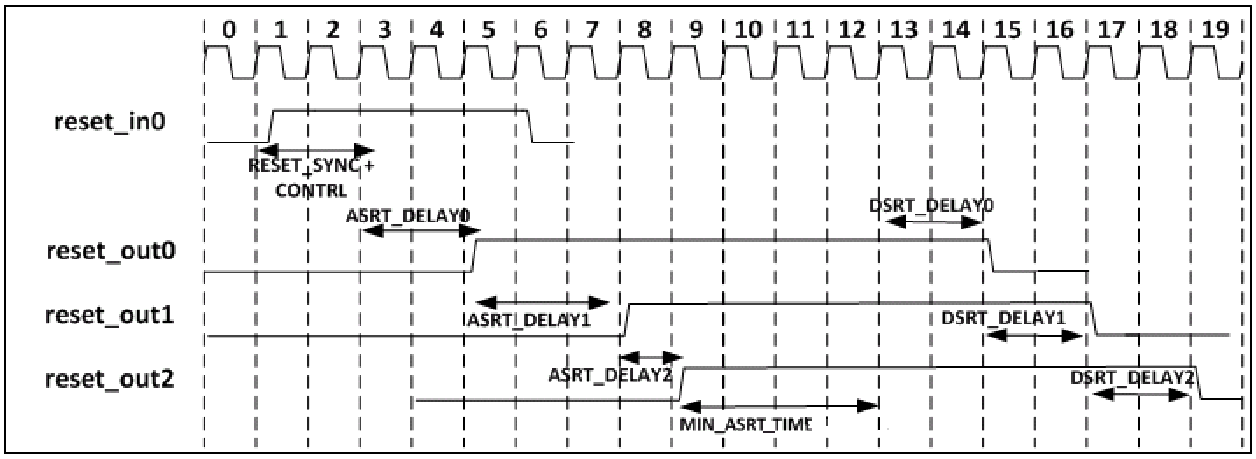
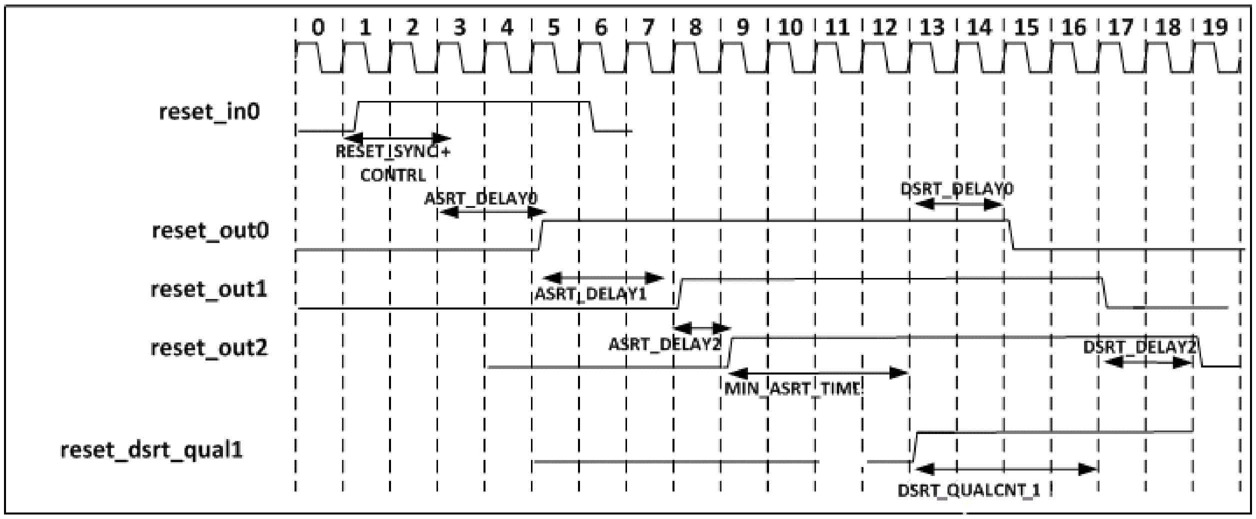


Figure 7-32: Sequencing with USE_DSRT_QUAL Set



Reset Sequencer CSR Registers

The CSR registers on the reset sequencer provide the following functionality:

- **Supports reset logging**
  - Ability to identify which reset is asserted.
  - Ability to determine whether any reset is currently active.
- **Supports software triggered resets**
  - Ability to generate reset by writing to the register.
  - Ability to disable assertion or de-assertion sequence.
- **Supports software sequenced reset**
  - Ability for the software to fully control the assertion/de-assertion sequence by writing to registers and stepping through the sequence.
- **Support reset override**
  - Ability to assert a particular component reset through software.

Reset Sequencer Status Register Offset 0x00

The **Status** register contains bits that indicate the sources of resets that cause a reset.

You can clear bits by writing 1 to the bit location. The Reset Sequencer ignores writes to bits with a value of 0. If the sequencer is reset (power-on-reset), all bits are cleared, except the power on reset bit.

Table 7-34: Values for the Status Register at Offset 0x00

Bit	Attribute	Default	Description
31	RO	0	**Reset Active**—Indicates that the sequencer is currently active in reset sequence (assertion or de-assertion).
30	RW1C	0	**Reset Asserted and waiting for SW to proceed:**—Set when there is an active reset assertion, and the next sequence is waiting for the software to proceed. Only valid when the **Enable SW sequenced reset entry** option is turned on.
29	RW1C	0	**Reset De-asserted and waiting for SW to proceed:**—Set when there is an active reset de-assertion, and the next sequence is waiting for the software to proceed. Only valid when the **Enable SW sequenced reset bring up** option is turned on.
28:26	RO	0	Reserved.
25:16	RW1C	0	**Reset de-assertion input qualification signal reset_dsrt_qual [9:0] status**—Indicates that the reset de-assertion's input signal qualification signal is set. This bit is set on the detection of assertion of the signal.
15:12	RO	0	Reserved.
11	RW1C	0	**reset_in9 was triggered**—Indicates that `resetin9` triggered the reset. Cleared by software by writing 1 to this bit location.

Bit	Attribute	Default	Description
10	RW1C	0	**reset_in8 was triggered**—Indicates that `reset_in8` triggered the reset. Cleared by software by writing a1 to this bit location.
9	RW1C	0	**reset_in7 was triggered**—Indicates that `reset_in7` triggered the reset. Cleared by software by writing 1 to this bit location.
8	RW1C	0	**reset_in6 was triggered**—Indicates that `reset_in6` triggered the reset. Cleared by software by writing 1 to this bit location.
7	RW1C	0	**reset_in5 was triggered**—Indicates that `reset_in5` triggered the reset. Cleared by software by writing 1 to this bit location.
6	RW1C	0	**reset_in4 was triggered**—Indicates that `reset_in4` triggered the reset. Cleared by software by writing 1 to this bit location.
5	RW1C	0	**reset_in3 was triggered**—Indicates that `reset_in3` triggered the reset. Cleared by software by writing 1 to this bit location.
4	RW1C	0	**reset_in2 was triggered**—Indicates that `reset_in2` triggered the reset. Cleared by software by writing 1 to this bit location.
3	RW1C	0	**reset_in1 was triggered**—Indicates that `reset_in1` triggered the reset. Cleared by software by writing 1 to this bit location.
2	RW1C	0	**reset_in0 was triggered**—Indicates that reset_in0 triggered. Cleared by software by writing 1 to this bit location.
1	RW1C	0	**Software triggered reset**—Indicates that the software triggered reset is set by the software, and triggering a reset.
0	RW1C	0	**Power-On-Reset was triggered**—Asserted whenever the reset to the sequencer is triggered. This bit is NOT reset when sequencer is reset. Cleared by software by writing 1 to this bit location.

Reset Sequencer Interrupt Enable Register Offset 0x04

The Interrupt Enable register contains the interrupt enable bit that you can use to enable any event triggering the IRQ of the reset sequencer.

Table 7-35: Values for the Interrupt Enable Register at Offset 0x04

Bit	Attribute	Default	Description
31	RO	0	Reserved.

Bit	Attribute	Default	Description
30	RW	0	**Interrupt on Reset Asserted and waiting for SW to proceed** enable. When set, the IRQ is set when the sequencer is waiting for the software to proceed in an assertion sequence.
29	RW	0	**Interrupt on Reset De-asserted and waiting for SW to proceed** enable. When set, the IRQ is set when the sequencer is waiting for the software to proceed in a de-assertion sequence.
28:26	RO	0	Reserved.
25:16	RW	0	**Interrupt on Reset de-assertion input qualification signal reset_dsrt_qual_ [9:0] status**— When set, the IRQ is set when the `reset_dsrt_qual[9:0]` status bit (per bit enable) is set.
15:12	RO	0	Reserved.
11	RW	0	**Interrupt on reset_in9 Enable**—When set, the IRQ is set when the `reset_in9` trigger status bit is set.
10	RW	0	**Interrupt on reset_in8 Enable**—When set, the IRQ is set when the `reset_in8` trigger status bit is set.
9	RW	0	**Interrupt on reset_in7 Enable**—When set, the IRQ is set when the `reset_in7` trigger status bit is set.
8	RW	0	**Interrupt on reset_in6 Enable**—When set, the IRQ is set when the `reset_in6` trigger status bit is set.
7	RW	0	**Interrupt on reset_in5 Enable**—When set, the IRQ is set when the `reset_in5` trigger status bit is set.
6	RW	0	**Interrupt on reset_in4 Enable**—When set, the IRQ is set when the `reset_in4` trigger status bit is set.
5	RW	0	**Interrupt on reset_in3 Enable**—When set, the IRQ is set when the `reset_in3` trigger status bit is set.
4	RW	0	**Interrupt on reset_in2 Enable**—When set, the IRQ is set when the `reset_in2` trigger status bit is set.
3	RW	0	**Interrupt on reset_in1 Enable**—When set, the IRQ is set when the `reset_in1` trigger status bit is set.
2	RW	0	**Interrupt on reset_in0 Enable**—When set, the IRQ is set when the `reset_in0` trigger status bit is set.

Bit	Attribute	Default	Description
1	RW	0	**Interrupt on Software triggered reset Enable**—When set, the IRQ is set when the software triggered reset status bit is set.
0	RW	0	**Interrupt on Power-On-Reset Enable**—When set, the IRQ is set when the power-on-reset status bit is set.

Reset Sequencer Control Register Offset 0x08

The Control register contains registers that you can use to control the reset sequencer.

Table 7-36: Values for the Control Register at Offset 0x08

Bit	Attribute	Default	Description
31:3	RO	0	Reserved.
2	RW	0	**Enable SW sequenced reset entry**—Enable a software sequenced reset entry sequence. Timer delays and input qualification are ignored, and only the software can sequence the entry.
1	RW	0	**Enable SW sequenced reset bring up**—Enable a software sequenced reset bring up sequence. Timer delays and input qualification are ignored, and only the software can sequence the bring up.
0	WO	0	**Initiate Reset Sequence**—Reset Sequencer writes this bit to 1 a single time in order to trigger the hardware sequenced warm reset. Reset Sequencer verifies that **Reset Active** is 0 before setting this bit, and always reads the value 0. To monitor this sequence, verify that **Reset Active** is asserted, and then subsequently de-asserted.

Reset Sequencer Software Sequenced Reset Entry Control Register Offset 0x0C

You can program the Reset Sequencer Software Sequenced Reset Entry Control register to control the reset entry sequence of the sequencer.

When the corresponding enable bit is set, the sequencer stops when the desired reset asserts, and then sets the **Reset Asserted and waiting for SW to proceed** bit. The Reset Sequencer proceeds only after the **Reset Asserted and waiting for SW to proceed** bit is cleared.

Table 7-37: Values for the Reset Sequencer Software Sequenced Reset Entry Controls Register at Offset 0x0C

Bit	Attribute	Default	Description
31:10	RO	0	Reserved.

Bit	Attribute	Default	Description
9:0	RW	3FF	**Per-reset SW sequenced reset entry enable**—This is a per-bit enable for SW sequenced reset entry. If `bitN` of this register is set, the sequencer sets the `bit30` of the status register when a `resetN` is asserted. It then waits for the `bit30` of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).

Reset Sequencer Software Sequenced Reset Bring Up Control Register Offset 0x10

You can program the Software Sequenced Reset Bring Up Control register to control the reset bring up sequence of the sequencer.

When the corresponding enable bit is set, the sequencer stops when the desired reset asserts, and then sets the **Reset De-asserted and waiting for SW to proceed** bit. The Reset Sequencer proceeds only after the **Reset De-asserted and waiting for SW to proceed** bit is cleared..

Table 7-38: Values for the Reset Sequencer Software Sequenced Bring Up Control Register at Offset 0x10

Bit	Attribute	Default	Description
31:10	RO	0	Reserved.
9:0	RW	3FF	**Per-reset SW sequenced reset entry enable**—This is a per-bit enable for SW sequenced reset bring up. If `bitN` of this register is set, the sequencer sets `bit29` of the status register when a `resetN` is asserted. It then waits for the `bit29` of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).

Reset Sequencer Software Direct Controlled Resets Offset 0x14

You can write a bit to 1 to assert the `reset_outN` signal, and to 0 to de-assert the `reset_outN` signal.

Table 7-39: Values for the Software Direct Controlled Resets at Offset 0x14

Bit	Attribute	Default	Description
31:26	RO	0	Reserved.
25:16	WO	0	**Reset Overwrite Trigger Enable** —This is a per-bit control trigger bit for the overwrite value to take effect.
15:10	RO	0	Reserved.
9:0	WO	0	**reset_outN Reset Overwrite Value**—This is a per-bit control of the `reset_out` bit. The Reset Sequencer can use this to forcefully drive the reset to a specific value. A value of 1 sets the `reset_out`. A value of 0 clears the `reset_out`. A write to this register only takes effect if the corresponding trigger bit in this register is set.

Reset Sequencer Software Reset Masking Offset 0x18

You can write a bit to 1 to assert the `reset_outN` signal, and to 0 to de-assert the `reset_outN` signal.

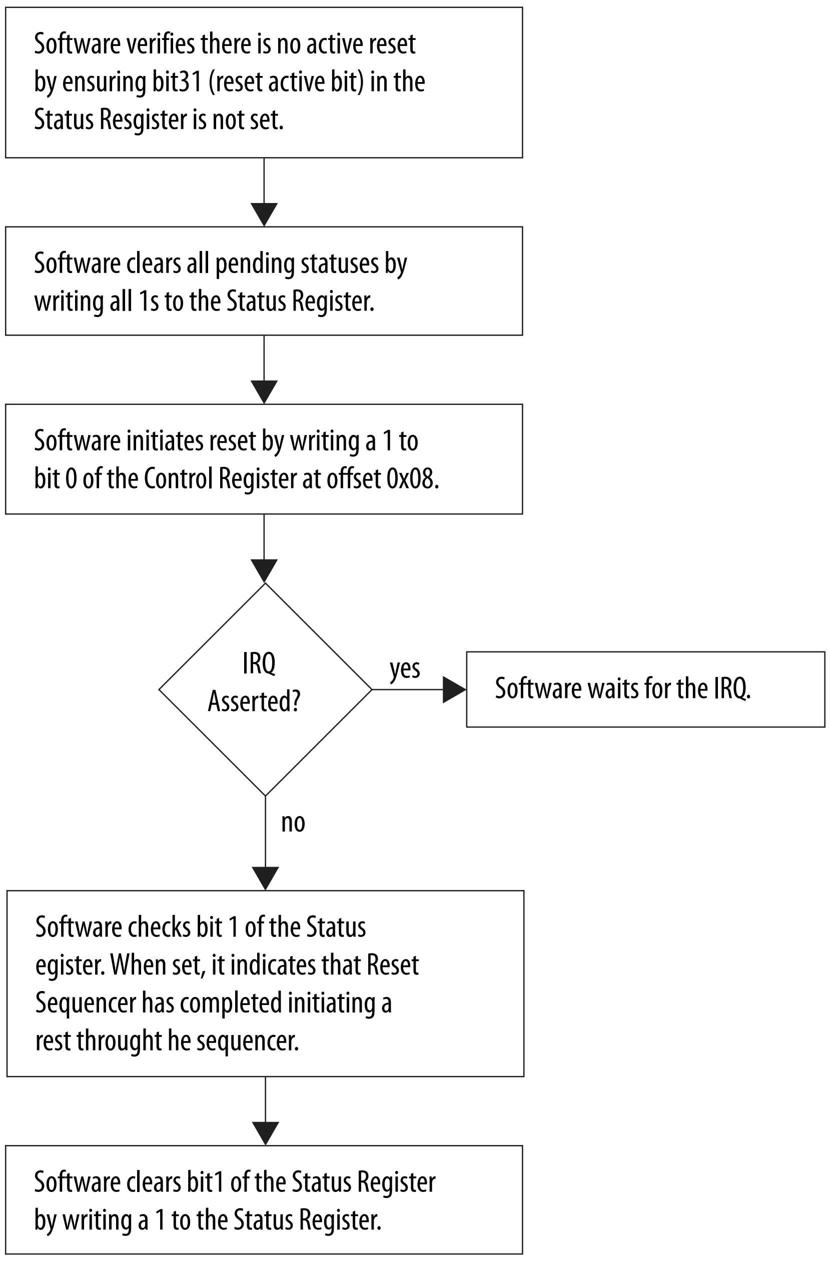
Table 7-40: Values for the Reset Sequencer Software Reset Masking at Offset 0x18

Bit	Attribute	Default	Description
31:10	RO	0	Reserved.
9:0	RW	0	**reset_outN "Reset Mask Enable"**—This is a per-bit control to mask the `reset_outN` bit. The Software Reset Masking masks the reset bit from being asserted during a reset assertion sequence. If the `reset_out` is already asserted, it does not de-assert the reset.

Reset Sequencer Software Flows

Reset Sequencer (Software-Triggered) Flow

Figure 7-33: Reset Sequencer (Software-Triggered) Flow


Software verifies there is no active reset by ensuring bit31 (reset active bit) in the Status Resgister is not set.
Software clears all pending statuses by writing all 1s to the Status Register.
Software initiates reset by writing a 1 to bit 0 of the Control Register at offset 0x08.
IRQ Asserted?
yes
Software waits for the IRQ.
no
Software checks bit 1 of the Status egister. When set, it indicates that Reset Sequencer has completed initiating a rest throught he sequencer.
Software clears bit1 of the Status Register by writing a 1 to the Status Register.

Reset Entry Flow

The following flow sequence occurs for a Reset Entry Flow:

- A reset is triggered either by the software, or when input resets to the Reset Sequencer are asserted.
- The IRQ is asserted, if the IRQ is enabled.
- Software reads the Status register to determine what reset was triggered.

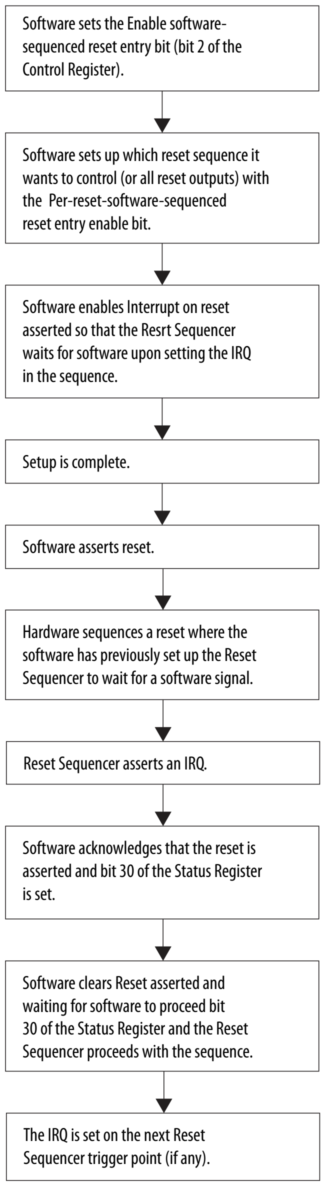
Reset Bring-Up Flow

The following flow sequence occurs for a Reset Bring-Up Flow:

- When a reset source is de-asserted, or when the reset entry sequence has completed without any more pending resets asserted, the bring-up flow is initiated.
- The IRQ is asserted, if the IRQ is enabled.
- Software reads the Status register to determine what reset was triggered.

Reset Entry (Software-Sequenced) Flow

Figure 7-34: Reset Entry (Software-Sequenced) Flow


Software sets the Enable software-sequenced reset entry bit (bit 2 of the Control Register).
Software sets up which reset sequence it wants to control (or all reset outputs) with the Per-reset-software-sequenced reset entry enable bit.
Software enables Interrupt on reset asserted so that the Resrt Sequencer waits for software upon setting the IRQ in the sequence.
Setup is complete.
Software asserts reset.
Hardware sequences a reset where the software has previously set up the Reset Sequencer to wait for a software signal.
Reset Sequencer asserts an IRQ.
Software acknowledges that the reset is asserted and bit 30 of the Status Register is set.
Software clears Reset asserted and waiting for software to proceed bit 30 of the Status Register and the Reset Sequencer proceeds with the sequence.
The IRQ is set on the next Reset Sequencer trigger point (if any).

Reset Bring-Up (Software-Sequenced) Flow

The sequence and flow is similar to the **Reset Entry (SW Sequenced)** flow, though, this flow uses the **reset bring-up** registers/bits in place of the **reset entry** registers/bits.

Related Information

Reset Entry (Software-Sequenced) Flow on page 7-66

Conduits

You can use the conduit interface type for interfaces that do not fit any of the other interface types, and to group any arbitrary collection of signals. Like other interface types, you can export or connect conduit interfaces. The *PCI Express-to-Ethernet* example in *Creating a System with Qsys* is an example of using a conduit interface for export. You can declare an associated clock interface for conduit interfaces in the same way as memory-mapped interfaces with the `associatedClock`.

To connect two conduit interfaces inside Qsys, the following conditions must be met:

- The interfaces must match exactly with the same signal roles and widths.
- The interfaces must be the opposite directions.
- Clocked conduit connections must have matching `associatedClocks` on each of their endpoint interfaces.

Note: To connect a conduit output to more than one input conduit interface, you can create a custom component. The custom component could have one input that connects to two outputs, and you can use this component between other conduits that you want to connect. For information about the Avalon Conduit interface, refer to the *Avalon Interface Specifications*

Related Information

Avalon Interface Specifications

Creating a System with Qsys on page 5-1

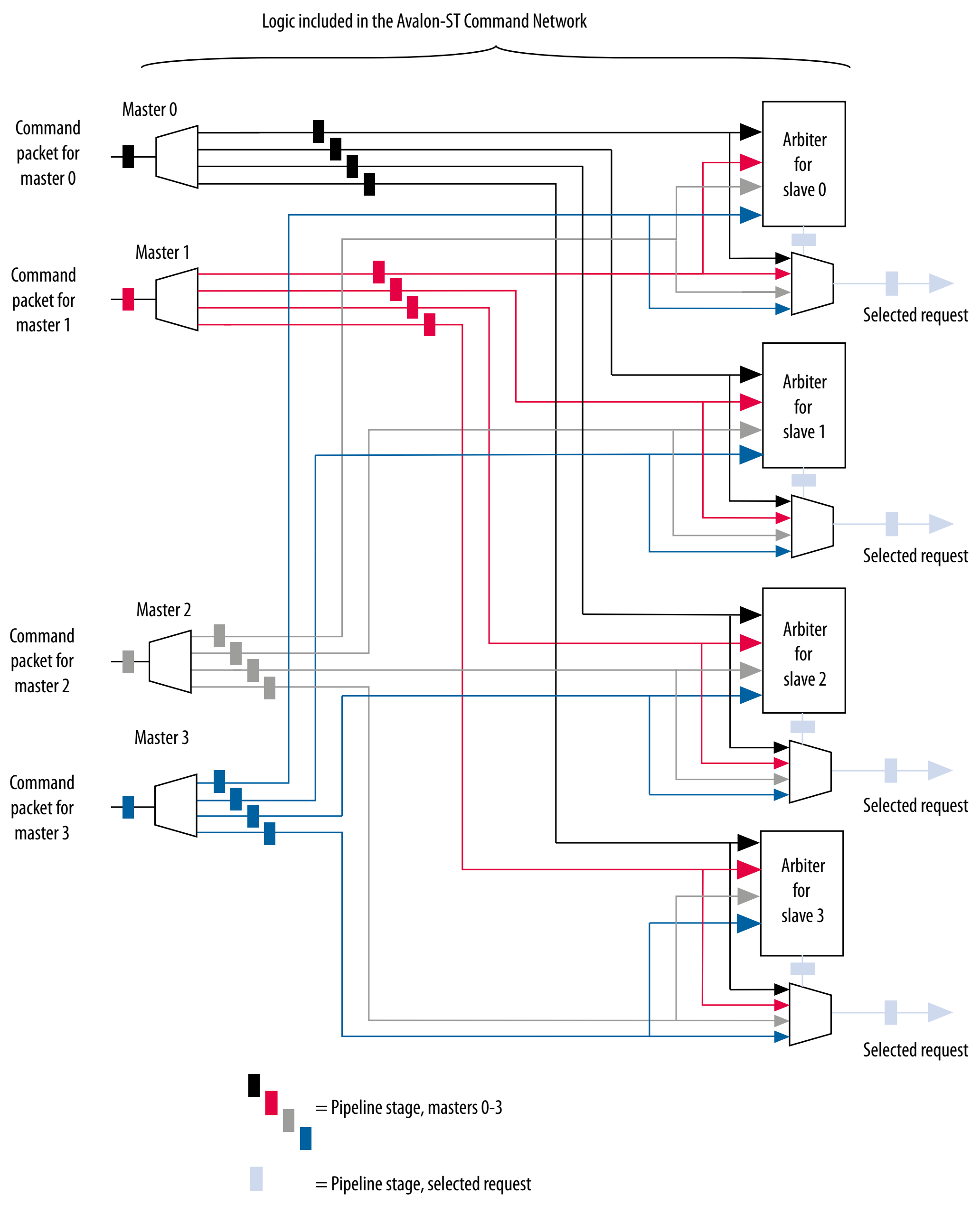
Interconnect Pipelining

If you set the **Limit interconnect pipeline stages to** parameter to a value greater than 0 on the **Project Settings** tab, Qsys automatically inserts Avalon-ST pipeline stages when you generate your design. The pipeline stages increase the f_{MAX} of your design by reducing the combinational logic depth. The cost is additional latency and logic.

The insertion of pipeline stages depends upon the existence of certain interconnect components. For example, in a single-slave system, no multiplexer exists; therefore multiplexer pipelining does not occur. In an extreme case, of a single-master to single-slave system, no pipelining occurs, regardless of the value of the**Limit interconnect pipeline stages to** option.

Figure 7-35: Pipeline Placement in Arbitration Logic

The example below shows the possible placement of up to four potential pipeline stages, which could be, before the input to the demultiplexer, at the output of the multiplexer, between the arbiter and the multiplexer, and at the outputs of the demultiplexer.


Logic included in the Avalon-ST Command Network
Master 0
Command packet for master 0
Master 1
Command packet for master 1
Master 2
Command packet for master 2
Master 3
Command packet for master 3
Arbiter for slave 0
Arbiter for slave 1
Arbiter for slave 2
Arbiter for slave 3
Selected request
Selected request
Selected request
Selected request
= Pipeline stage, masters 0-3
= Pipeline stage, selected request

Note: For more information about manually inserting and removing pipelines from your system, refer to *Creating a System With Qsys*. Refer to *Optimizing Qsys System Performance* for more information about pipelined Avalon-MM Interfaces.

Related Information

- Creating a System With Qsys on page 5-1

Manually Controlling Pipelining in the Qsys Interconnect

The **Memory-Mapped Interconnect** tab allows you to manipulate pipleline connections in the Qsys interconnect. You access the **Memory-Mapped Interconnect** tab by clicking the **Show System With Qsys Interconnect** command on the System menu.

Note: To increase interconnect frequency, you should first try increasing the value of the **Limit interconnect pipeline stages to** option on the **Interconnect Requirements** tab. You should only consider manually pipelining the interconnect if changes to this option do not improve frequency, and you have tried all other options to achieve timing closure, including the use of a bridge. Manually pipelining the interconnect should only be applied to complete systems.

1. In the **Interconnect Requirements** tab, first try increasing the value of the **Limit interconnect pipeline stages to** option until it no longer gives significant improvements in frequency, or until it causes unacceptable effects on other parts of the system.
2. In the Quartus II software, compile your design and run timing analysis.
3. Using the timing report, identify the critical path through the interconnect and determine the approximate mid-point. The following is an example of a timing report:

```
2.800 0.000 cpu_instruction_master|out_shifter[63]|q
3.004 0.204 mm_domain_0|addr_router_001|Equal5~0|datac
3.246 0.242 mm_domain_0|addr_router_001|Equal5~0|combout
3.346 0.100 mm_domain_0|addr_router_001|Equal5~1|dataa
3.685 0.339 mm_domain_0|addr_router_001|Equal5~1|combout
4.153 0.468 mm_domain_0|addr_router_001|src_channel[5]~0|datad
4.373 0.220 mm_domain_0|addr_router_001|src_channel[5]~0|combout
```

4. In Qsys, click **System** > **Show System With Qsys Interconnect**.
5. In the **Memory-Mapped Interconnect** tab, select the interconnect module that contains the critical path. You can determine the name of the module from the hierarchical node names in the timing report.
6. Click **Show Pipelinable Locations**. Qsys display all possible pipeline locations in the interconnect. Right-click the possible pipeline location to insert or remove a pipeline stage.
7. Locate the possible pipeline location that is closest to the mid-point of the critical path. The names of the blocks in the memory-mapped interconnect tab correspond to the module instance names in the timing report.
8. Right-click the location where you want to insert a pipeline, and then click **Insert Pipeline**.
9. Regenerate the Qsys system, recompile the design, and then rerun timing analysis. If necessary, repeat the manual pipelining process again until timing requirements are met.

Manual pipelining has the following limitations:

- If you make changes to your original system's connectivity after manually pipelining an interconnect, your inserted pipelines may become invalid. Qsys displays warning messages when you generate your system if invalid pipeline stages are detected. You can remove invalid pipeline stages with the **Remove Stale Pipelines** option in the **Memory-Mapped Interconnect** tab. Altera recommends that you do not make changes to the system's connectivity after manual pipeline insertion.
- Review manually-inserted pipelines when upgrading to newer versions of Qsys. Manually-inserted pipelines in one version of Qsys may not be valid in a future version.

Related Information

Specify Qsys $system Interconnect Requirements

Qsys System Design Components on page 10-1

Error Correction Coding (ECC) in Qsys Interconnect

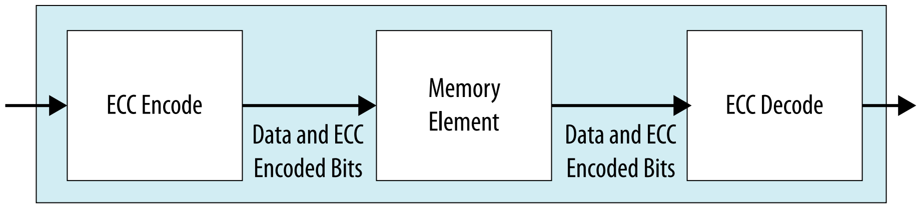
Error Correction Coding (ECC) allows the Qsys interconnect to detect and correct errors in order to improve data integrity in memory blocks.

As transistors become smaller, computer hardware is more susceptible to data corruption. Data corruption causes Single Event Upsets (SEUs) and increases the probability of Failures in Time (FIT) rates in computer systems. SEU events without error notification can cause the system to be stuck in an unknown response state, and increase the probability of FIT rates.

ECC encodes the data bus with a Hamming code before it writes it to the memory device, and then decodes and performs error checking on the data on output.

Note: Qsys sends uncorrectable errors in memeory elements as a `DECERR` on the response bus. This feature is currently only supported for `rdata_FIFO` instances when back pressure occurs on the `wait_request` signal.

Figure 7-36: High-Level Implementation of RDATA FIFO with ECC Enabled



Related Information

- **Read and Write Responses** on page 7-27
- **Specify Qsys Interconnect Requirements**

AMBA 3 AXI Protocol Specification Support (version 1.0)

Qsys allows memory-mapped connections between AXI3 components, AXI3 and AXI4 components, and AXI3 and Avalon interfaces with some unique or exceptional support.

Refer to the *AMBA 3 Protocol Specifications* on the ARM website for more information.

Related Information

AMBA 3 Protocol Specifications

Channels

Qsys 14.0 has the following support and restrictions for AXI3 channels.

Read and Write Address Channels

All signals are allowed with some limitations.

The following limitations are present in Qsys 14.0:

- Supports 64-bit addressing.
- ID width limited to 18-bits.
- HPS-FPGA master interface has a 12-bit ID.

Write Data, Write Response, and Read Data Channels

All signals are allowed with some limitations.

The following limitations are present in Qsys 14.0:

- Data widths limited to a maximum of 1024-bits
- Limited to a fixed byte width of 8-bits

Low Power Channel

Low power extensions are not supported in Qsys, version 14.0.

Cache Support

`AWCACHE` and `ARCACHE` are passed to an AXI slave unmodified.

Bufferable

Qsys interconnect treats AXI transactions as non-bufferable. All responses must come from the terminal slave.

When connecting to Avalon-MM slaves, since they do not have write responses, the following exceptions apply:

- For Avalon-MM slaves, the write response are generated by the slave agent once the write transaction is accepted by the slave. The following limitation exists for an Avalon bridge:
- For an Avalon bridge, the response is generated before the write reaches the endpoint; users must be aware of this limitation and avoid multiple paths past the bridge to any endpoint slave, or only perform bufferable transactions to an Avalon bridge.

Cacheable (Modifiable)

Qsys interconnect acknowledges the cacheable (modifiable) attribute of AXI transactions.

It does not change the address, burst length, or burst size of non-modifiable transactions, with the following exceptions:

- Qsys considers a wide transaction to a narrow slave as modifiable because the size requires reduction.
- Qsys may consider AXI read and write transactions as modifiable when the destination is an Avalon slave. The AXI transaction may be split into multiple Avalon transactions if the slave is unable to accept the transaction. This may occur because of burst lengths, narrow sizes, or burst types.

Qsys ignores all other bits, for example, read allocate or write allocate because the interconnect does not perform caching. By default, Qsys considers Avalon master transactions as non-bufferable and non-cacheable, with the allocate bits tied low. Qsys provides compile-time options to control the cache behavior of Avalon transactions on a per-master basis.

Security Support

TrustZone refers to the security extension of the ARM architecture, which includes the concept of "secure" and "non-secure" transactions, and a protocol for processing between the designations.

The interconnect passes the `AWPROT` and `ARPROT` signals to the endpoint slave without modification. It does not use or modify the `PROT` bits.

Refer to *Creating a System with Qsys* for more information about secure systems and the TrustZone feature.

Related Information

- **Creating a System with Qsys** on page 5-1

Atomic Accesses

Exclusive accesses are supported for AXI slaves by passing the lock, transaction ID, and response signals from master to slave, with the limitation that slaves that do not reorder responses. Avalon slaves do not support exclusive accesses, and always return `OKAY` as a response. Locked accesses are also not supported.

Response Signaling

Full response signaling is supported. Avalon slaves always return `OKAY` as a response.

Ordering Model

Qsys interconnect provides responses in the same order as the commands are issued.

To prevent reordering, for slaves that accept reordering depths greater than 0, Qsys does not transfer the transaction ID from the master, but provides a constant transaction ID of 0. For slaves that do not reorder, Qsys allows the transaction ID to be transferred to the slave. To avoid cyclic dependencies, Qsys supports a single outstanding slave scheme for both reads and writes. Changing the targeted slave before all responses have returned stalls the master, regardless of transaction ID.

AXI and Avalon Ordering

According to the *AMBA Protocol Specifications*, there is no ordering requirement between reads and writes. However, Avalon has an implicit ordering model that requires transactions from a master to the same slave to be in order. As a result, there is a potential read-after-write risk when Avalon masters transact to AXI slaves. In response to this potential risk, Avalon interfaces provide a compile-time option to enforce strict order. When turned on, the Avalon interface waits for outstanding write responses before issuing reads.

Data Buses

Narrow bus transfers are supported. AXI write strobes can have any pattern that is compatible with the address and size information. Altera recommends that transactions to Avalon slaves follow Avalon `byteenable` limitations for maximum compatibility.

Note: Byte `0` is always bits `[7:0]` in the interconnect, following AXI's and Avalon's byte (address) invariance scheme.

Unaligned Address Commands

Unaligned address commands are commands with addresses that do not conform to the data width of a slave. Since Avalon-MM slaves accept only aligned addresses, Qsys modifies unaligned commands from AXI masters to the correct data width. Qsys must preserve commands issued by AXI masters when passing the commands to AXI slaves.

Note: Unaligned transfers are aligned if downsizing occurs. For example, when downsizing to a bus width narrower than that required by the transaction size, `AWSIZE` or `ARSIZE`, the transaction must be modified.

Avalon and AXI Transaction Support

Qsys 14.0 supports transactions between Avalon and interfaces with some limitations.

Transaction Cannot Cross 4KB Boundaries

When an Avalon master issues a transaction to an AXI slave, the transaction cannot cross 4KB boundaries. Non-bursting Avalon masters already follow this boundary restriction.

Handling Read Side Effects

Read side effects can occur when more bytes than necessary are read from the slave, and the unwanted data that are read are later inaccessible on subsequent reads. For write commands, the correct byteenable paths are asserted based on the size of the transactions. For read commands, narrow-sized bursts are broken up into multiple non-bursting commands, and each command with the correct byteenable paths asserted.

Note: Qsys always assumes that the byteenable is asserted based on the size of the command, not the address of the command. The following scenarios are examples:

- For a 32-bit AXI master that issues a read command with an unaligned address starting at address `0x01`, and a burstcount of 2 to a 32-bit Avalon slave, the starting address is: `0x00`.
- For a 32-bit AXI master that issues a read command with an unaligned address starting at address `0x01`, with 4-bytes to an 8-bit AXI slave, the starting address is: `0x00`.

AMBA 3 APB Protocol Specification Support (version 1.0)

AMBA APB provides a low-cost interface that is optimized for minimal power consumption and reduced interface complexity. You can use AMBA APB to interface to peripherals which are low-bandwidth and do not require the high performance of a pipelined bus interface. Signal transitions are sampled at the rising edge of the clock to enable the integration of APB peripherals easily into any design flow.

Qsys allows connections between APB components, and AXI3, AXI4, and Avalon memory-mapped interfaces. The following sections describe unique or exceptional APB support in the Qsys software.

Refer to the *AMBA APB Protocol Specifications* for AXI4 on the ARM website for more information.

Related Information

AMBA APB Protocol Specifications

Bridges

With APB, you cannot use bridge components that use multiple `PSELx` in Qsys. As a workaround, you can group PSELx, and then send the packet to the slave directly.

Altera recommends as an alternative that you instantiate the APB bridge and all the APB slaves in Qsys. You should then connect the slave side of the bridge to any high speed interface and connect the master side of the bridge to the APB slaves. Qsys creates the interconnect on either side of the APB bridge and creates only one `PSEL` signal.

Alternatively, you can connect a bridge to the APB bus outside of Qsys. Use an Avalon/AXI bridge to export the Avalon/AXI master to the top-level, and then connect this Avalon/AXI interface to the slave side of the APB bridge. Alternatively, instantiate the APB bridge in Qsys and export APB master to the top- level, and from there connect to APB bus outside of Qsys.

Burst Adaptation

APB is a non-bursting interface. Therefore, for any AXI or Avalon master with bursting support, a burst adapter is inserted before the slave interface and the burst transaction is translated into a series of non-bursting transactions before reaching the APB slave.

Width Adaptation

Qsys allows different data width connections with APB. When connecting a wider master to a narrower APB slave, the width adapter converts the wider transactions to a narrower transaction to fit the APB slave data width. APB does not support Write Strobe. Therefore, when you connect a narrower transaction to a wider APB slave, the slave cannot determine which byte lane to write. In this case, the slave data may be overwritten or corrupted.

Error Response

Error responses are returned to the master. Qsys performs error mapping if the master is an AXI3 or AXI4 master, for example, `RRESP/BRESP= SLVERR`. For the case when the slave does not use `SLVERR` signal, an `OKAY` response is sent back to master by default.

AMBA AXI4 Memory-Mapped Interface Support (version 2.0)

Qsys allows memory-mapped connections between AXI4 components, AXI4 and AXI3 components, and AXI4 and Avalon interfaces with some unique or exceptional support.

Burst Support

Qsys supports `INCR` bursts up to 256 beats. Qsys converts long bursts to multiple bursts in a packet with each burst having a length less than or equal to `MAX_BURST` when going to AXI3 or Avalon slaves.

For narrow-sized transfers, bursts with Avalon slaves as destinations are shortened to multiple non-bursting transactions in order to transmit the correct address to the slaves, since Avalon slaves always perform full-sized `datawidth` transactions.

Bursts with AXI3 slaves as destinations are shortened to multiple bursts, with each burst length less than or equal to 16. Bursts with AXI4 slaves as destinations are not shortened.

QoS

Qsys routes 4-bit QoS signals (Quality of Service Signaling) on the read and write address channels directly from the master to the slave.

Transactions from AXI3 and Avalon masters have a default value of `4'b0000`, which indicates that the transactions are not part of the QoS flow. QoS values are not used for slaves that do not support QoS.

For Qsys 14.0, there are no programmable QoS registers or compile-time QoS options for a master that overrides its real or default value.

Regions

For Qsys 14.0, there is no support for the optional regions feature. AXI4 slaves with `AXREGION` signals are allowed. `AXREGION` signals are driven with the default value of `0x0`, and are limited to one entry in a master's address map.

Write Response Dependency

Write response dependency as specified in the *AMBA Protocol Specifications* for AXI4 is not supported.

Related Information

AMBA Protocol Specifications

AWCACHE and ARCACHE

For AXI4, Qsys meets the requirement for modifiable and non-modifiable transactions. The modifiable bit refers to `ARCACHE[1]`and `AWCACHE[1]`.

Width Adaptation and Data Packing in Qsys

Data packing applies only to systems where the data width of masters is less than the data width of slaves.

The following rules apply:

- Data packing is supported when masters and slaves are Avalon-MM.
- Data packing is not supported when any master or slave is an AXI3, AXI4, or APB component.

For example, for a read/write command with a 32-bit master connected to a 64-bit slave, and a transaction of 2 burstcounts, Qsys sends 2 separate read/write commands to access the 64-bit data width of the slave. Data packing is only supported if the system does not contain AXI3, AXI4, or APB masters or slaves.

Ordering Model

Out of order support is not implemented in Qsys, version 14.0. Qsys processes AXI slaves as device non-bufferable memory types.

The following describes the required behavior for the device non-bufferable memory type:

- Write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- Transaction characteristics must not be modified.
- Reads must not be pre-fetched. Writes must not be merged.
- Non-modifiable read and write transactions.

(`AWCACHE[1] = 0 or ARCACHE[1] = 0`) from the same ID to the same slave must remain ordered. The interconnect always provides responses in the same order as the commands issued. Slaves that support reordering provide a constant transaction ID to prevent reordering. AXI slaves that do not reorder are provided with transaction IDs, which allows exclusive accesses to be used for such slaves.

Read and Write Allocate

Read and write allocate does not apply to Qsys interconnect, which does not have caching features, and always receives responses from an endpoint.

Locked Transactions

Locked transactions are not supported for Qsys, version 14.0.

Memory Types

For AXI4, Qsys processes transactions as though the endpoint is a device memory type. For device memory types, using non-bufferable transactions to force previous bufferable transactions to finish is irrelevant, because Qsys interconnect always identifies transactions as being non-bufferable.

Mismatched Attributes

There are rules for how multiple masters issue cache values to a shared memory region. The interconnect meets requirements as long as cache signals are not modified.

Signals

Qsys supports up to 64-bits for the `BUSER`, `WUSER` and `RUSER` sideband signals. AXI4 allows some signals to be omitted from interfaces by aligning them with the default values as defined in the *AMBA Protocol Specifications* on the ARM website.

Related Information

AMBA Protocol Specifications

AMBA AXI4 Streaming Interface Support (version 1.0)

Connection Points

Qsys allows you to connect an AXI4 stream interface to another AXI4 stream interface.

The connection is point-to-point without adaptation and must be between an `axi4stream_master` and `axi4stream_slave`. Connected interfaces must have the same port roles and widths.

Non matching master to slave connections, and multiple masters to multiple slaves connections are not supported.

AXI4 Streaming Connection Point Parameters

Table 7-41: AXI4 Streaming Connection Point Parameters

Name	Type	Description
`associatedClock`	string	Name of associated clock interface.

Name	Type	Description
`associatedReset`	string	Name of associated reset interface

AXI4 Streaming Connection Point Signals

Table 7-42: AXI4 Stream Connection Point Signals

Port Role	Width	Master Direction	Slave Direction	Required
`tvalid`	1	Output	Input	Yes
`tready`	1	Input	Output	No
`tdata`[6]	8:4096	Output	Input	No
`tstrb`	1:512	Output	Input	No
`tkeep`	1:512	Output	Input	No
`tid`[7]	1:8	Output	Input	No
`tdest`[8]	1:4	Output	Input	No
`tuser`[9]	1:4096	Output	Input	No
`tlast`	1	Output	Input	No

Adaptation

AXI4 stream adaptation support is not available. AXI4 stream master and slave interface signals and widths must match.

AMBA AXI4-Lite Protocol Specification Support (version 2.0)

AXI4-Lite is a sub-set of AMBA AXI4. It is suitable for simpler control register-style interfaces that do not require the full functionality of AXI4.

Qsys 14.0 supports the following AXI4-Lite features:

- Transactions with a burst length of 1.
- Data accesses use the full width of a data bus (32- bit or 64-bit) for data accesses, and no narrow-size transactions.
- Non-modifiable and non-bufferable accesses.
- No exclusive accesses.

AXI4-Lite Signals

Qsys supports all AXI4-Lite interface signals. All signals are required.

(6) integer in mutiple of bytes
(7) maximum 8-bits
(8) maximum 4-bits
(9) number of bits in multiple of the number of bytes of `tdata`

Table 7-43: AXI4-Lite Signals

Global	Write Address Channel	Write Data Channel	Write Response Channel	Read Address Channel	Read Data Channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

AXI4-Lite Bus Width

AXI4-Lite masters or slaves must have either 32-bit or 64-bit bus widths. Qsys interconnect inserts a width adapter if a master and slave pair have different widths.

AXI4-Lite Outstanding Transactions

AXI-Lite supports outstanding transactions. The options to control outstanding transactions is set in the parameter editor for the selected component.

AXI4-Lite IDs

AXI4-Lite does not support IDs. Qsys performs ID reflection inside the slave agent.

Connections Between AXI3/4 and AXI4-Lite

AXI4-Lite Slave Requirements

For an AXI4-Lite slave side, the master can be any master interface type, such as an Avalon (with bursting), AXI3, or AXI4. Qsys allows the following connections and inserts adapters, if needed.

- **Burst adapter**—Avalon and AXI3 and AXI4 bursting masters require a burst adapter to shorten the burst length to 1 before sending a transaction to an AXI4-Lite slave.
- Qsys interconnect uses a width adapter for mismatched data widths.
- Qsys interconnect performs ID reflection inside the slave agent.
- An AXI4-Lite slave must have an address width of at least 12-bits.
- AXI4-Lite does not have the AXSIZE parameter. Narrow master to a wide AXI4-Lite slave is not supported. For masters that support narrow-sized bursts, for example, AXI3 and AXI4, a burst to an AXI4-Lite slave must have a burst size equal to or greater than the slave's burst size.

AXI4-Lite Data Packing

Qsys interconnect does not support AXI4-Lite data packing.

AXI4-Lite Response Merging

When Qsys interconnect merges SLVERR and DECERR, the error responses are not sticky. The response is based on priority and the master always sees a DECERR. When SLVERR and DECERR are merged, it is based on their priorities, not stickiness. DECERR receives priority in this case, even if SLVERR returns first.

Port Roles (Interface Signal Types)

Each interfaces defines a number of signal roles and their behavior. Many signal roles are optional, allowing IP component designers the flexibility to select only the signal roles necessary to implement the required functionality.

AXI Master Interface Signal Types

Table 7-44: AXI Master Interface Signal Types

Name	Direction	Width
`araddr`	output	1 - 64
`arburst`	output	2
`arcache`	output	4
`arid`	output	1 - 18
`arlen`	output	4
`arlock`	output	2
`arprot`	output	3
`arready`	input	1
`arsize`	output	3
`aruser`	output	1 - 64
`arvalid`	output	1
`awaddr`	output	1 - 64
`awburst`	output	2
`awcache`	output	4
`awid`	output	1 - 18
`awlen`	output	4
`awlock`	output	2
`awprot`	output	3
`awready`	input	1
`awsize`	output	3
`awuser`	output	1 - 64
`awvalid`	output	1
`bid`	input	1 - 18
`bready`	output	1

Name	Direction	Width
`bresp`	input	2
`bvalid`	input	1
`rdata`	input	8, 16, 32, 64, 128, 256, 512, 1024
`rid`	input	1 - 18
`rlast`	input	1
`rready`	output	1
`rresp`	input	2
`rvalid`	input	1
`wdata`	output	8, 16, 32, 64, 128, 256, 512, 1024
`wid`	output	1 - 18
`wlast`	output	1
`wready`	input	1
`wstrb`	output	1, 2, 4, 8, 16, 32, 64, 128
`wvalid`	output	1

AXI Slave Interface Signal Types

Table 7-45: AXI Slave Interface Signal Types

Name	Direction	Width
`araddr`	input	1 - 64
`arburst`	input	2
`arcache`	input	4
`arid`	input	1 - 18
`arlen`	input	4
`arlock`	input	2
`arprot`	input	3
`arready`	output	1
`arsize`	input	3
`aruser`	input	1 - 64
`arvalid`	input	1
`awaddr`	input	1 - 64
`awburst`	input	2

Name	Direction	Width
awcache	input	4
awid	input	1 - 18
awlen	input	4
awlock	input	2
awprot	input	3
awready	output	1
awsize	input	3
awuser	input	1 - 64
awvalid	input	1
bid	output	1 - 18
bready	input	1
bresp	output	2
bvalid	output	1
rdata	output	8, 16, 32, 64, 128, 256, 512, 1024
rid	output	1 - 18
rlast	output	1
rready	input	1
rresp	output	2
rvalid	output	1
wdata	input	8, 16, 32, 64, 128, 256, 512, 1024
wid	input	1 - 18
wlast	input	1
wready	output	1
wstrb	input	1, 2, 4, 8, 16, 32, 64, 128
wvalid	input	1

AXI4 Master Interface Signal Types

Table 7-46: AXI4 Master Interface Signal Types

Name	Direction	Width
araddr	output	1 - 64
arburst	output	2

Name	Direction	Width
arcache	output	4
arid	output	1 - 18
arlen	output	8
arlock	output	1
arprot	output	3
arready	input	1
arregion	output	1 - 4
arsize	output	3
aruser	output	1 - 64
arvalid	output	1
awaddr	output	1 - 64
awburst	output	2
awcache	output	4
awid	output	1 - 18
awlen	output	8
awlock	output	1
awprot	output	3
awqos	output	1 - 4
awready	input	1
awregion	output	1 - 4
awsize	output	3
awuser	output	1 - 64
awvalid	output	1
bid	input	1 - 18
bready	output	1
bresp	input	2
buser	input	1 - 64
bvalid	input	1
rdata	input	8, 16, 32, 64, 128, 256, 512, 1024
rid	input	1 - 18
rlast	input	1

Name	Direction	Width
rready	output	1
rresp	input	2
ruser	input	1 - 64
rvalid	input	1
wdata	output	8, 16, 32, 64, 128, 256, 512, 1024
wid	output	1 - 18
wlast	output	1
wready	input	1
wstrb	output	1, 2, 4, 8, 16, 32, 64, 128
wuser	output	1 - 64
wvalid	output	1

AXI4 Slave Interface Signal Types

Table 7-47: AXI4 Slave Interface Signal Types

Name	Direction	Width
araddr	input	1 - 64
arburst	input	2
arcache	input	4
arid	input	1 - 18
arlen	input	8
arlock	input	1
arprot	input	3
arqos	input	1 - 4
arready	output	1
arregion	input	1 - 4
arsize	input	3
aruser	input	1 - 64
arvalid	input	1
awaddr	input	1 - 64
awburst	input	2
awcache	input	4

Name	Direction	Width
awid	input	1 - 18
awlen	input	8
awlock	input	1
awprot	input	3
awqos	input	1 - 4
awready	output	1
awregion	inout	1 - 4
awsize	input	3
awuser	input	1 - 64
awvalid	input	1
bid	output	1 - 18
bready	input	1
bresp	output	2
bvalid	output	1
rdata	output	8, 16, 32, 64, 128, 256, 512, 1024
rid	output	1 - 18
rlast	output	1
rready	input	1
rresp	output	2
ruser	output	1 - 64
rvalid	output	1
wdata	input	8, 16, 32, 64, 128, 256, 512, 1024
wlast	input	1
wready	output	1
wstrb	input	1, 2, 4, 8, 16, 32, 64, 128
wuser	input	1 - 64
wvalid	input	1

AXI4 Stream Master and Slave Interface Signal Types

Table 7-48: AXI4 Stream Master and Slave Interface Signal Types

Name	Width	Master Direction	Slave Direction	Required
`tvalid`	1	Output	Input	Yes
`tready`	1	Input	Output	No
`tdata`	8:4096	Output	Input	No
`tstrb`	1:512	Output	Input	No
`tkeep`	1:512	Output	Input	No
`tid`	1:8	Output	Input	No
`tdest`	1:4	Output	Input	No
`tuser`	1	Output	Input	No
`tlast`	1:4096	Output	Input	No

APB Interface Signal Types

Table 7-49: APB Interface Signal Types

Name	Width	Direction APB Master	Direction APB Slave	Required
`paddr`	[1:32]	output	input	yes
`psel`	[1:16]	output	input	yes
`penable`	1	output	input	yes
`pwrite`	1	output	input	yes
`pwdata`	[1:32]	output	input	yes
`prdata`	[1:32]	input	output	yes
`pslverr`	1	input	output	no
`pready`	1	input	output	yes
`paddr31`	1	output	input	no

Avalon Memory-Mapped Interface Signal Roles

The following table lists signal roles for the Avalon-MM interface. Signal roles allow you to create masters that use bursts for read and write processing. When necessary, Qsys interconnect enables the connection between the master and slave pair. When the master and slave interfaces match, a direct connection is possible.

This specification does not require all signals to exist in an Avalon-MM interface. There is no one signal that is always required. The minimum requirements for an Avalon-MM interface are `readdata` for a read-only interface, or `writedata` and `write` for a write-only interface.

Table 7-50: Avalon-MM Signal Roles

Avalon-MM signals can be active high, or active low. When active low, the signal name ends with _n.

Signal Role	Width	Direction	Description
		Fundamental Signals	
`address`	1 - 64	Master → Slave	Masters: By default, the `address` signal represents a byte address. The value of the address must be aligned to the data width. To write to specific bytes within a data word, the master must use the `byteenable` signal. Refer to the `addressUnits` interface property for word addressing. Slaves: By default, the interconnect translates the byte address into a word address in the slave's address space. Each slave access is for a word of data from the perspective of the slave. For example, `address = 0`= 0 selects the first word of the slave. `address = 1` selects the second word of the slave. Refer to the `addressUnits` interface property for byte addressing.

`byteenable`
 	
`byteenable_n`
	2, 4, 8, 16, 32, 64, 128	Master → Slave	Enables specific byte lane(s) during transfers on interfaces of width greater than 8 bits. Each bit in `byteenable` corresponds to a byte in `writedata` and `readdata`. The master bit <n> of `byteenable` indicates whether byte <n> is being written to. During writes, `byteenables` specify which bytes are being written to. Other bytes should be ignored by the slave. During reads, `byteenables` indicate which bytes the master is reading. Slaves that simply return `readdata` with no side effects are free to ignore `byteenables` during reads. If an interface does not have a `byteenable` signal, the transfer proceeds as if all `byteenables` are asserted.
			
			When more than one bit of the `byteenable` signal is asserted, all asserted lanes are adjacent. The number of adjacent lines must be a power of 2. The specified bytes must be aligned on an address boundary for the size of the data. For

Signal Role	Width	Direction	Description
			example, the following values are legal for a 32-bit slave: • 1111 writes full 32 bits • 0011 writes lower 2 bytes • 1100 writes upper 2 bytes • 0001 writes byte 0 only • 0010 writes byte 1 only • 0100 writes byte 2 only • 1000 writes byte 3 only To avoid unintended side effects, Altera strongly recommends that you use the `byteenable` signal in systems with different word sizes.
`debugaccess`	1	Master → Slave	When asserted, allows the Nios II processor to write on-chip memories configured as ROMs.
`read` `read_n`	1	Master → Slave	Asserted to indicate a `read` transfer. If present, `readdata` is required.
`readdata`	8,16, 32, 64,128, 256, 512, 1024	Slave → Master	The `readdata` driven from the slave to the master in response to a `read` transfer.

Signal Role	Width	Direction	Description
`response [1:0]`	2	Slave → Master	The `response` signal is optional signal that carries the response status. **Note:** Because the signal is shared, an interface cannot issue or accept a write response and a read response in the same clock cycle. • `00: OKAY`—Successful response for a transaction. • `01: RESERVED`—Encoding is reserved. • `10: SLAVEERROR`—Error from an endpoint slave. Indicates an unsuccessful transaction. • `11: DECODEERROR`—Indicates attempted access to an undefined location. For read responses: • Interconnect sends one response with each `readdata`. A read burst length of `N` results in `N` responses. It is not valid to produce fewer responses, even in the event of an error. It is valid for the response signal value to be different for each `readdata` in the burst. • The interface must have read control signals. Pipeline support is possible with the `readdatavalid` signal. • On read errors, the corresponding `readdata` is "don't care". For write responses: • The interface must have write control signals. Qsys completes all write commands with write responses if the write signal is present. The interface must have a `writeresponsevalid` signal. • Qsys sends one response for each write command. A write burst results in only one response, which the interconnect must send after the final write transfer in the burst is accepted.
`write` `write_n`	1	Master → Slave	Asserted to indicate a `write` transfer. If present, `writedata` is required.
`writedata`	8,16, 32, 64, 128, 256, 512, 1024	Master → Slave	Data for write transfers. The width must be the same as the width of `readdata` if both are present.
Wait-State Signals			
`lock`	1	Master → Slave	`lock` ensures that once a master wins arbitration, it maintains access to the slave for

Signal Role	Width	Direction	Description
			multiple transactions. It is asserted coincident with the first `read` or `write` of a locked sequence of transactions. It is deasserted on the final transaction of a locked sequence of transactions. `lock` assertion does not guarantee that arbitration will be won. After the lock-asserting master has been granted, it retains grant until it is deasserted. A master equipped with `lock` cannot be a burst master. Arbitration priority values for lock-equipped masters are ignored. `lock` is particularly useful for read-modify-write (RMW) operations. The typical read-modify-write operation includes the following steps: **1.** Master A asserts lock and reads 32-bit data that has multiple bit fields. **2.** Master A deasserts lock, changes one bit field, and writes the 32-bit data back. `lock` prevents master B from performing a write between Master A's read and write.

`waitrequest`
 	
`waitrequest_n`
	1	Slave → Master	Asserted by the slave when it is unable to respond to a `read` or `write` request. Forces the master to wait until the interconnect is ready to proceed with the transfer. At the start of all transfers, a master initiates the transfer and waits until `waitrequest` is deasserted. A master must make no assumption about the assertion state of `waitrequest` when the master is idle: `waitrequest` may be high or low, depending on system properties.
			
			When `waitrequest` is asserted, master control signals to the slave are to remain constant with the exception of `beginbursttransfer`. For a timing diagram illustrating the `beginbursttransfer` signal, refer to the figure in *Read Bursts*.
			
			An Avalon-MM slave may assert `waitrequest` during idle cycles. An Avalon-MM master may initiate a transaction when `waitrequest` is asserted and wait for that signal to be deasserted. To avoid system lockup, a slave device should assert `waitrequest` when in reset.

Pipeline Signals			

Signal Role	Width	Direction	Description

`readdatavalid`
 	
`readdatavalid_n`
	1	Slave → Master	Used for variable-latency, pipelined `read` transfers. When asserted, indicates that the `readdata` signal contains valid data. A slave with `readdatavalid` must assert this signal for one cycle for each `read` access received. There must be at least one cycle of latency between acceptance of the `read` and assertion of `readdatavalid`. For a timing diagram illustrating the `readdatavalid` signal, refer to *Pipelined Read Transfer with Variable Latency*.
			
			A slave may assert `readdatavalid` to transfer data to the master independently of whether or not the slave is stalling a new command with `waitrequest`.
			
			Required if the master supports pipelined reads. Bursting masters with read functionality must include the `readdatavalid` signal.

`writeresponse-valid`			An optional signal. If present, the interface issues write responses for write commands. When asserted, the value on the response signal is a valid write response. `Writeresponsevalid` is only asserted one clock cycle or more after the write command is accepted. There is at least a one clock cycle latency from command acceptance to assertion of `writeresponsevalid`.
Burst Signals			
`burstcount`	1 – 11	Master → Slave	Used by bursting masters to indicate the number of transfers in each burst. The value of the maximum `burstcount` parameter must be a power of 2. A burstcount interface of width <n> can encode a max burst of size $2^{(<n>-1)}$. For example, a 4-bit `burstcount` signal can support a maximum burst count of 8. The minimum `burstcount` is 1. The `constantBurstBehavior` property controls the timing of the `burstcount` signal. Bursting masters with read functionality must include the `readdatavalid` signal. For bursting masters and slaves using byte addresses, the following restriction applies to the width of the address: *`<address_w> >= <burstcount_w>`* `+log`$_2$*`(<symbols_per_word_of_interface>).`* For bursting masters and slaves using word addresses, the `log`$_2$ term above is omitted.

Signal Role	Width	Direction	Description
`beginburst-transfer`	1	Interconnect → Slave	Asserted for the first cycle of a burst to indicate when a burst transfer is starting. This signal is deasserted after one cycle regardless of the value of `waitrequest`. For a timing diagram illustrating `beginbursttransfer`, refer to the figure in *Read Bursts*. `beginbursttransfer` is optional. A slave can always internally calculate the start of the next write burst transaction by counting data transfers. Altera recommends that you **do not** use this signal. This signal exists to support legacy memory controllers.

Related Information

- Read and Write Responses
- Avalon Interface Specifications

Avalon Streaming Interface Signal Roles

Each signal in an Avalon-ST source or sink interface corresponds to one Avalon-ST signal role. An Avalon-ST interface may contain only one instance of each signal role. All Avalon-ST signal roles apply to both sources and sinks and have the same meaning for both.

Table 7-51: Avalon-ST Interface Signals

In the following table, all signal roles are active high.

Signal Role	Width	Direction	Description
		Fundamental Signals	
`channel`	1 – 128	Source → Sink	The `channel` number for data being transferred on the current cycle. If an interface supports the channel signal, it must also define the `maxChannel` parameter.
`data`	1 – 4,096	Source → Sink	The `data` signal from the source to the sink, typically carries the bulk of the information being transferred. The contents and format of the `data` signal is further defined by parameters.
`error`	1 – 256	Source → Sink	A bit mask used to mark errors affecting the data being transferred in the current cycle. A single bit in `error` is used for each of the errors recognized by the component, as defined by the `errorDe-scriptor` property.

Signal Role	Width	Direction	Description
`ready`	1	Sink → Source	Asserted high to indicate that the sink can accept data. `ready` is asserted by the sink on cycle <n> to mark cycle <*n* + `readyLatency`> as a ready cycle. The source may only assert `valid` and transfer data during `ready` cycles. Sources without a `ready` input cannot be backpressured. Sinks without a `ready` output never need to backpressure.
`valid`	1	Source → Sink	Asserted by the source to qualify all other source to sink signals. The sink samples data other source-to-sink signals on ready cycles where `valid` is asserted. All other cycles are ignored. Sources without a `valid` output implicitly provide valid data on every cycle that they are not being backpressured. Sinks without a `valid` input expect valid data on every cycle that they are not backpressuring.
Packet Transfer Signals			
`empty`	1 – 8	Source → Sink	Indicates the number of symbols that are empty, that is, do not represent valid data. The empty signal is not used on interfaces where there is one symbol per beat.
`endofpacket`	1	Source → Sink	Asserted by the source to mark the end of a packet.
`startofpacket`	1	Source → Sink	Asserted by the source to mark the beginning of a packet.

Avalon Clock Source Signal Roles

An Avalon Clock source interface drives a clock signal out of a component.

Table 7-52: Clock Source Signal Roles

Signal Role	Width	Direction	Required	Description
`clk`	1	Output	Yes	An output clock signal.

Avalon Clock Sink Signal Roles

A clock sink provides a timing reference for other interfaces and internal logic.

Table 7-53: Clock Sink Signal Roles

Signal Role	Width	Direction	Required	Description
`clk`	1	Input	Yes	A clock signal. Provides synchronization for internal logic and for other interfaces.

Avalon Conduit Signal Roles

Table 7-54: Conduit Signal Roles

Signal Role	Width	Direction	Description
<any>	<*n*>	In, out, or bidirectional	A conduit interface consists of one or more input, output, or bidirectional signals of arbitrary width. Conduits can have any user-specified role. You can connect compatible Conduit interfaces inside a Qsys system provided the roles and widths match and the directions are opposite.

Avalon Tristate Conduit Signal Roles

The following table lists the signal defined for the Avalon Tristate Conduit interface. All Avalon-TC signals apply to both masters and slaves and have the same meaning for both

Table 7-55: Tristate Conduit Interface Signal Roles

Signal Role	Width	Direction	Required	Description
`request`	1	Master → Slave	Yes	The meaning of `request` depends on the state of the `grant` signal, as the following rules dictate. When `request` is asserted and `grant` is deasserted, `request` is requesting access for the current cycle. When `request` is asserted and `grant` is asserted, `request` is requesting access for the next cycle. Consequently, `request` should be deasserted on the final cycle of an access. The `request` is deasserted in the last cycle of a bus access. It can be reasserted immediately following the final cycle of a transfer. This protocol makes both rearbitration and continuous bus access possible if no other masters are requesting access. Once asserted, `request` must remain asserted until granted. Consequently, the shortest bus access is 2 cycles. Refer to *Tristate Conduit Arbitration Timing* for an example of arbitration timing.
`grant`	1	Slave → Master	Yes	When asserted, indicates that a tristate conduit master has been granted access to perform transactions. `grant` is asserted in response to the `request` signal. It remains asserted until 1 cycle following the deassertion of `request`.
`<name>_in`	1 – 1024	Slave → Master	No	The input signal of a logical tristate signal.

Signal Role	Width	Direction	Required	Description
`<name>_out`	1 – 1024	Master → Slave	No	The output signal of a logical tristate signal.
`<name>_outen`	1	Master → Slave	No	The output enable for a logical tristate signal.

Avalon Tri-State Slave Interface Signal Types

Table 7-56: Tri-state Slave Interface Signal Types

Name	Width	Direction	Required	Description
`address`	1 - 32	input	No	Address lines to the slave port. Specifies a byte offset into the slave's address space.
`read` `read_n`	1	input	No	Read-request signal. Not required if the slave port never outputs data. If present, data must also be used.
`write` `write_n`	1	input	No	Write-request signal. Not required if the slave port never receives data from a master. If present, data must also be present, and `writebyteen-able` cannot be present.
`chipselect` `chipselect_n`	1	input	No	When present, the slave port ignores all Avalon-MM signals unless `chipselect` is asserted. `chipselect` is always present in combina-tion with read or write
`outputenable` `outputenable_n`	1	input	Yes	Output-enable signal. When deasserted, a tri-state slave port must not drive its data lines otherwise data contention may occur.
`data`	8,16, 32, 64, 128, 256, 512, 1024	bidir	No	Bidirectional data. During write transfers, the FPGA drives the data lines. During read transfers the slave device drives the data lines, and the FPGA captures the data signals and provides them to the master.

Name	Width	Direction	Required	Description

`byteenable`
`byteenable_n`
	2, 4, 8,16, 32, 64, 128	input	No	Enables specific byte lane(s) during transfers.
				
				Each bit in byteenable corresponds to a byte lane in data. During writes, `byteenables` specify which bytes the master is writing to the slave. During reads, `byteenables` indicates which bytes the master is reading. Slaves that simply return data with no side effects are free to ignore `byteenables` during reads.
				
				When more than one byte lane is asserted, all asserted lanes are guaranteed to be adjacent. The number of adjacent lines must be a power of 2, and the specified bytes must be aligned on an address boundary for the size of the data. The are legal values for a 32-bit slave:
				
				`1111    writes full 32 bits`
				`0011    writes lower 2 bytes`
				`1100    writes upper 2 bytes`
				`0001    writes byte 0 only`
				`0010    writes byte 1 only`
				`0100    writes byte 2 only`
				`1000    writes byte 3 only`

`writebyteenable` `writebyteenable_n`	2,4,8,16, 32, 64,128	input	No	Equivalent to the logical AND of the `byteenable` and write signals. When used, the write signal is not used.
`begintransfer1`	1	input	No	Asserted for the first cycle of each transfer.

Note: All Avalon signals are active high. Avalon signals that can also be asserted low list both versions in the **Signal Role** column.

Avalon Interrupt Sender Signal Roles

Table 7-57: Interrupt Sender Signal Roles

Signal Role	Width	Direction	Required	Description
`irq` `irq_n`	1	Output	Yes	Interrupt Request. A slave asserts `irq` when it needs service. The interrupt receiver determines the relative priority of the interrupts.

Avalon Interrupt Receiver Signal Roles

Table 7-58: Interrupt Receiver Signal Roles

Signal Role	Width	Direction	Required	Description
`irq`	1–32	Input	Yes	`irq` is an <*n*>-bit vector, where each bit corresponds directly to one IRQ sender with no inherent assumption of priority.

Document Revision History

The table below indicates edits made to the *Qsys Interconnect* content since its creation.

Table 7-59: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• Fixed Priority Arbitration. • Added topic: *Read and Write Responses*. • Added topic: *Error Correction Coding (ECC) in Qsys Interconnect*. • Added: `response [1:0]`, *Avalon Memory-Mapped Interface Signal Roles*. • Added `writeresponsevalid`, *Avalon Memory-Mapped Interface Signal Roles*.
December 2014	14.1.0	• Read error responses, Avalon Memory-Mapped Interface Signal, `response`. • Burst Adapter Implementation Options: Generic converter (slower, lower area), Per-burst-type converter (faster, higher area).

Date	Version	Changes
August 2014	14.0a10.0	• Updated Qsys Packet Format for Memory-Mapped Master and Slave Interfaces table, `Protection`. • Streaming Interface renamed to Avalon Streaming Interfaces. • Added *Response Merging* under *Memory-Mapped Interfaces*.
June 2014	14.0.0	• AXI4-Lite support. • AXI4-Stream support. • Avalon-ST adapter parameters. • IRQ Bridge. • Handling Read Side Effects note added.
November 2013	13.1.0	• HSSI clock support. • Reset Sequencer. • Interconnect pipelining.
May 2013	13.0.0	• AMBA APB support. • Auto-inserted Avalon-ST adapters feature. • Moved Address Span Extender to the *Qsys System Design Components* chapter.
November 2012	12.1.0	• AMBA AXI4 support.
June 2012	12.0.0	• AMBA AXI3 support. • Avalon-ST adapters. • Address Span Extender.
November 2011	11.0.1	Template update.
May 2011	11.0.0	Removed beta status.
December 2010	10.1.0	Initial release.

Related Information

Quartus II Handbook Archive

Optimizing Qsys System Performance 8

2015.05.04

QII5V1 Subscribe Send Feedback

You can optimize system interconnect performance for Altera® designs that you create with the Qsys system integration tool.

The foundation of any system is the interconnect logic that connects hardware blocks or components. Creating interconnect logic is prone to errors, is time consuming to write, and is difficult to modify when design requirements change. The Qsys system integration tool addresses these issues and provides an automatically generated and optimized interconnect designed to satisfy your system requirements.

Qsys supports Avalon, AMBA AXI3 (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB3 (version 1.0) interface specifications.

Note: Recommended Altera practices may improve clock frequency, throughput, logic utilization, or power consumption of your Qsys design. When you design a Qsys system, use your knowledge of your design intent and goals to further optimize system performance beyond the automated optimization available in Qsys.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specifications
- Creating a System with Qsys on page 5-1
- Creating Qsys Components on page 6-1
- Qsys Interconnect on page 7-1

Designing with Avalon and AXI Interfaces

Qsys Avalon and AXI interconnect for memory-mapped interfaces is flexible, partial crossbar logic that connects master and slave interfaces.

Avalon Streaming (Avalon-ST) links connect point-to-point, unidirectional interfaces and are typically used in data stream applications. Each a pair of components is connected without any requirement to arbitrate between the data source and sink.

Because Qsys supports multiplexed memory-mapped and streaming connections, you can implement systems that use multiplexed logic for control and streaming for data in a single design.

Related Information

- Creating Qsys Components on page 6-1

Designing Streaming Components

When you design streaming component interfaces, you must consider integration and communication for each component in the system. One common consideration is buffering data internally to accommodate latency between components.

For example, if the component's Avalon-ST output or source of streaming data is back-pressured because the ready signal is de-asserted, then the component must back-pressure its input or sink interface to avoid overflow.

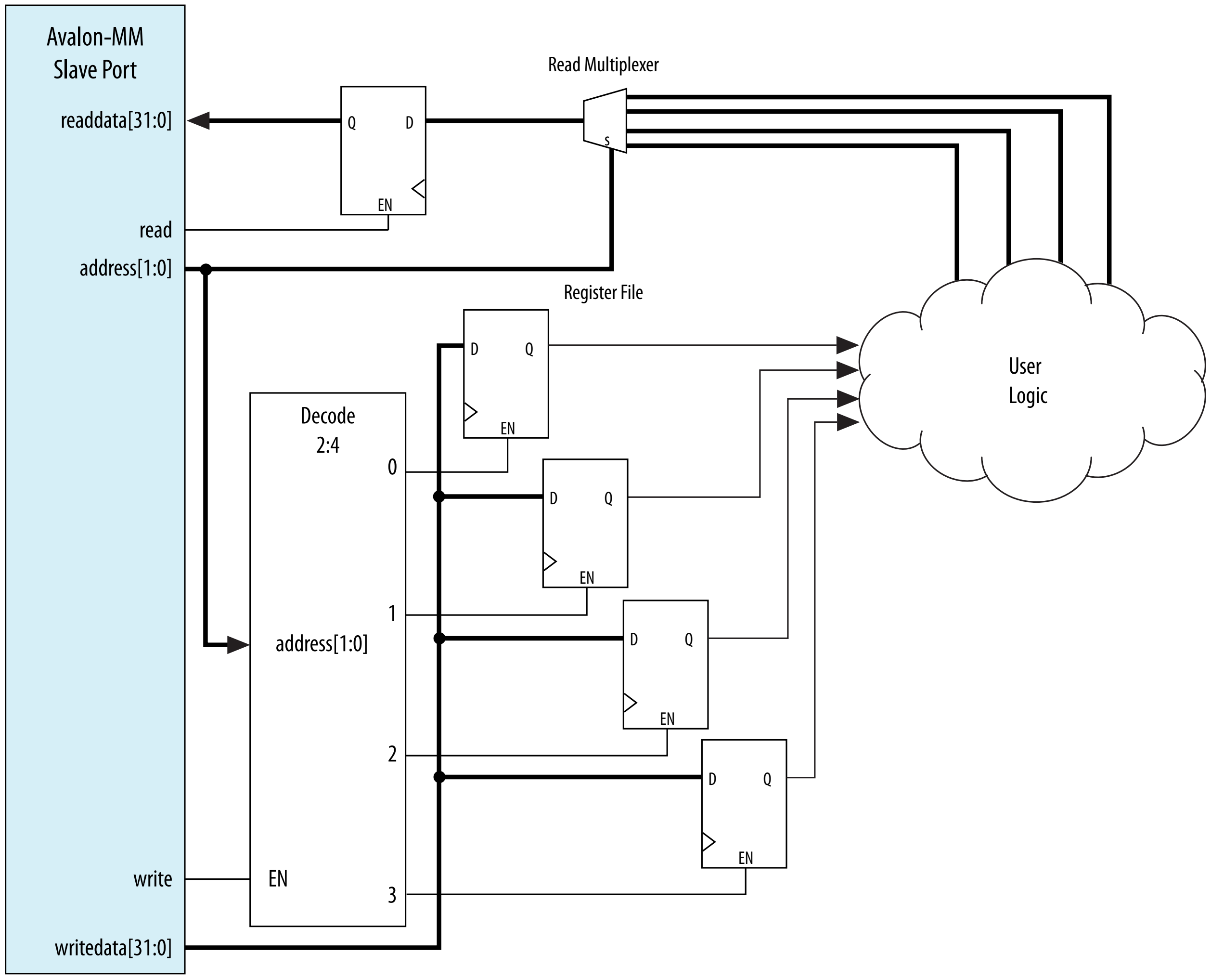
You can use a FIFO to back-pressure internally on the output side of the component so that the input can accept more data even if the output is back-pressured. Then, you can use the FIFO almost full flag to back-pressure the sink interface or input data when the FIFO has only enough space to satisfy the internal latency. You can drive the data valid signal of the output or source interface with the FIFO not empty flag when that data is available.

Designing Memory-Mapped Components

When designing with memory-mapped components, you can implement any component that contains multiple registers mapped to memory locations, for example, a set of four output registers to support software read back from logic. Components that implement read and write memory-mapped transactions require three main building blocks: an address decoder, a register file, and a read multiplexer.

The decoder enables the appropriate 32-bit or 64-bit register for writes. For reads, the address bits drive the multiplexer selection bits. The read signal registers the data from the multiplexer, adding a pipeline stage so that the component can achieve a higher clock frequency.

Figure 8-1: Control and Status Registers (CSR) in a Slave Component



This slave component has write wait states and one read wait state. Alternatively, if you want high throughput, you may set both the read and write wait states to zero, and then specify a read latency of one, because the component also supports pipelined reads.

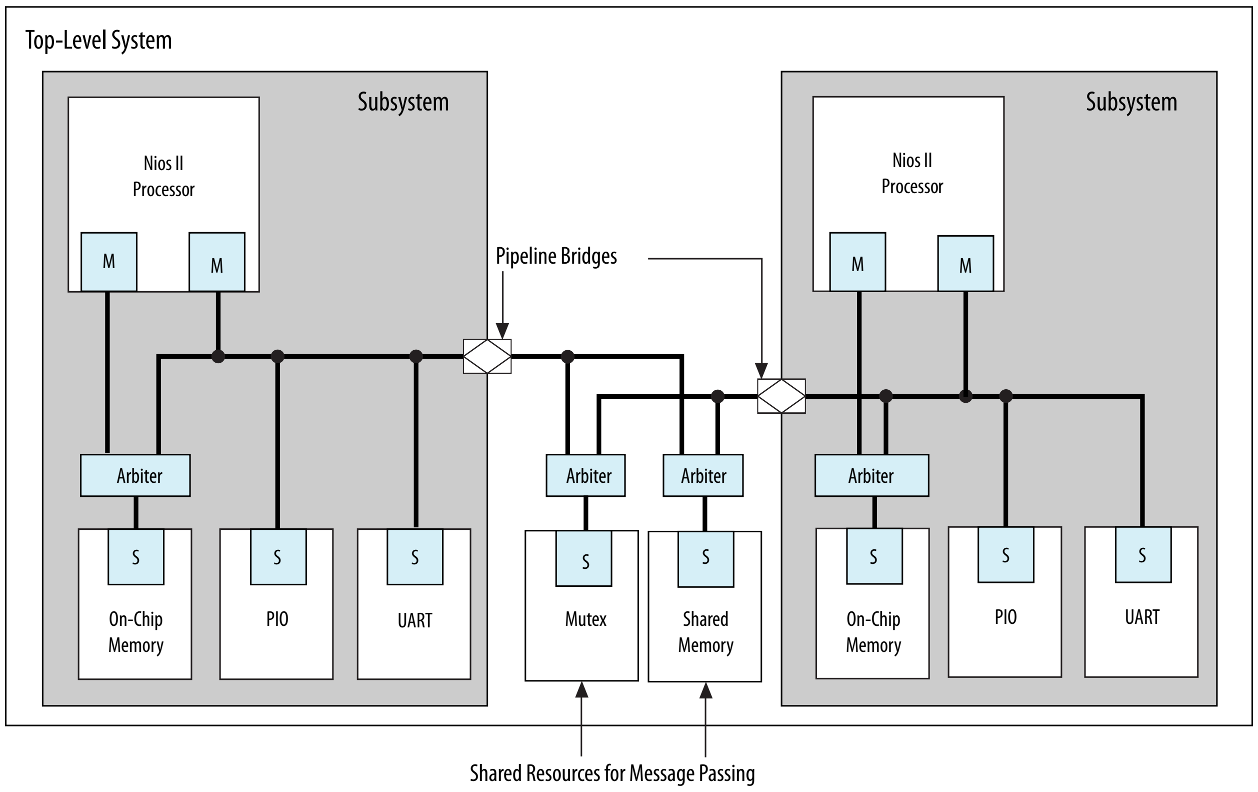
Using Hierarchy in Systems

You can use hierarchy to sub-divide a system into smaller subsystems that you can then connect in a top-level Qsys system. Additionally, If a design contains one or more identical functional units, the functional unit can be defined as a subsystem and instantiated multiple times within a top-level system.

Hierarchy can simplify verification control of slaves connected to each master in a memory-mapped system. Before you implement subsystems in your design, you should plan the system hierarchical blocks at the top-level, using the following guidelines:

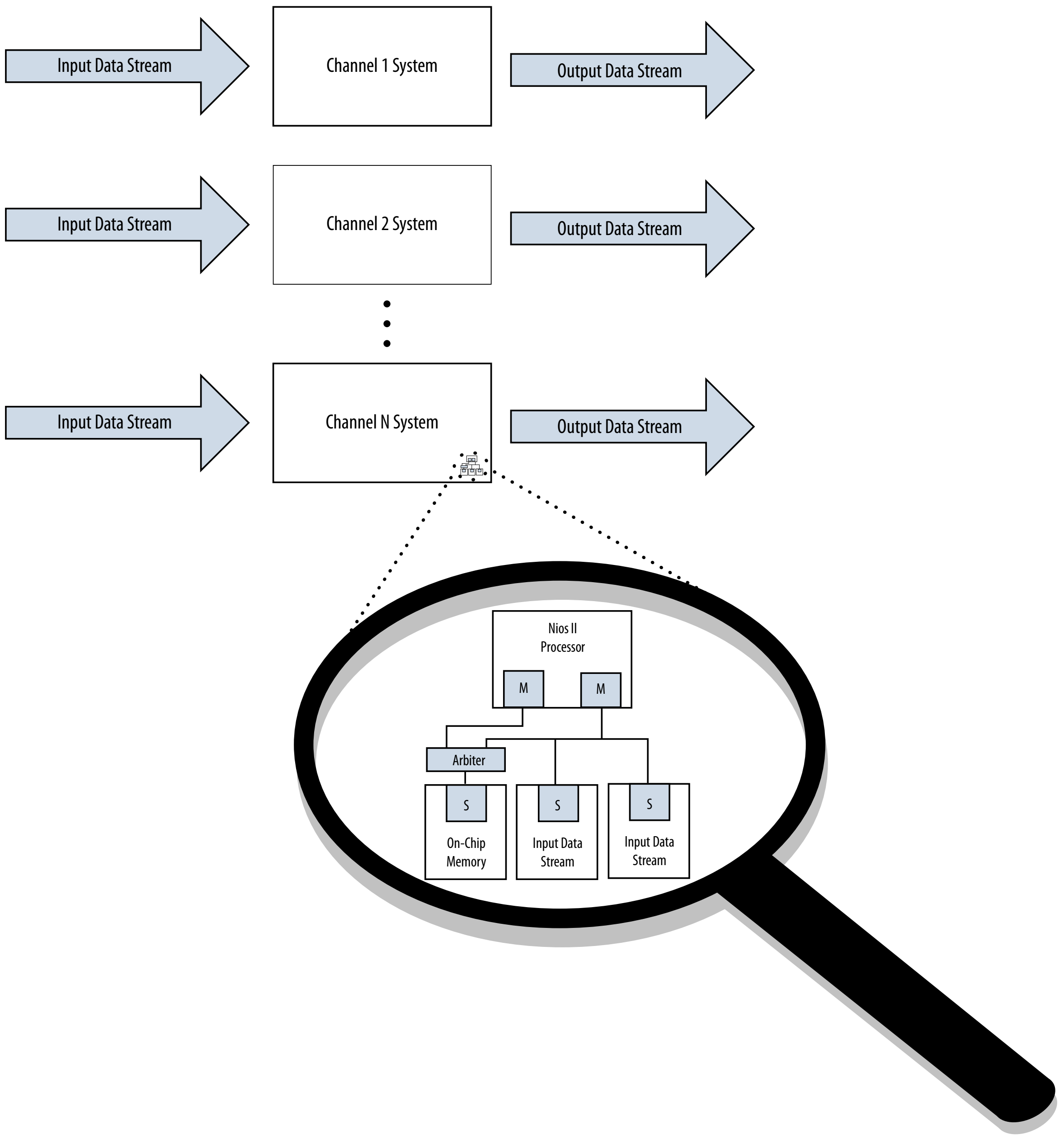
- **Plan shared resources**—Determine the best location for shared resources in the system hierarchy. For example, if two subsystems share resources, add the components that use those resources to a higher-level system for easy access.
- **Plan shared address space between subsystems**—Planning the address space ensures you can set appropriate sizes for bridges between subsystems.
- **Plan how much latency you may need to add to your system**—When you add a pipeline bridge between subsystems, you may add latency to the overall system. You can reduce the added latency by parameterizing the pipeline bridge with zero cycles of latency.

Figure 8-2: Passing Messages Between Subsystems



In this example, two Nios II processor subsystems share resources for message passing. Bridges in each subsystem export the Nios II data master to the top-level system that includes the mutex (mutual exclusion component) and shared memory component (which could be another on-chip RAM, or a controller for an off-chip RAM device).

Figure 8-3: Multi Channel System



You can also design systems that process multiple data channels by instantiating the same subsystem for each channel. This approach is easier to maintain than a larger, non-hierarchical system. Additionally, such systems are easier to scale because you can calculate the required resources as a multiple of the subsystem requirements.

Using Concurrency in Memory-Mapped Systems

Qsys interconnect uses parallel hardware in FPGAs, which allows you to design concurrency into your system and process transactions simultaneously.

Implementing Concurrency With Multiple Masters

Implementing concurrency requires multiple masters in a Qsys system. Systems that include a processor contain at least two master interfaces because the processors include separate instruction and data masters. You can catagorize master components as follows:

- General purpose processors, such as Nios II processors
- DMA (direct memory access) engines
- Communication interfaces, such as PCI Express

Because Qsys generates an interconnect with slave-side arbitration, every master interface in a system can issue transfers concurrently, as long as they are not posting transfers to the same slave. Concurrency is limited by the number of master interfaces sharing any particular slave interface. If a design requires higher data throughput, you can increase the number of master and slave interfaces to increase the number of transfers that occur simultaneously. The example below shows a system with three master interfaces.

Figure 8-4: Avalon Multiple Master Parallel Access

In this Avalon example, the DMA engine operates with Avalon-MM read and write masters. However, an AXI DMA interface typically has only one master, because in the AXI standard, the write and read channels on the master are independent and can process transactions simultaneously. The yellow lines represent active simultaneously connections.

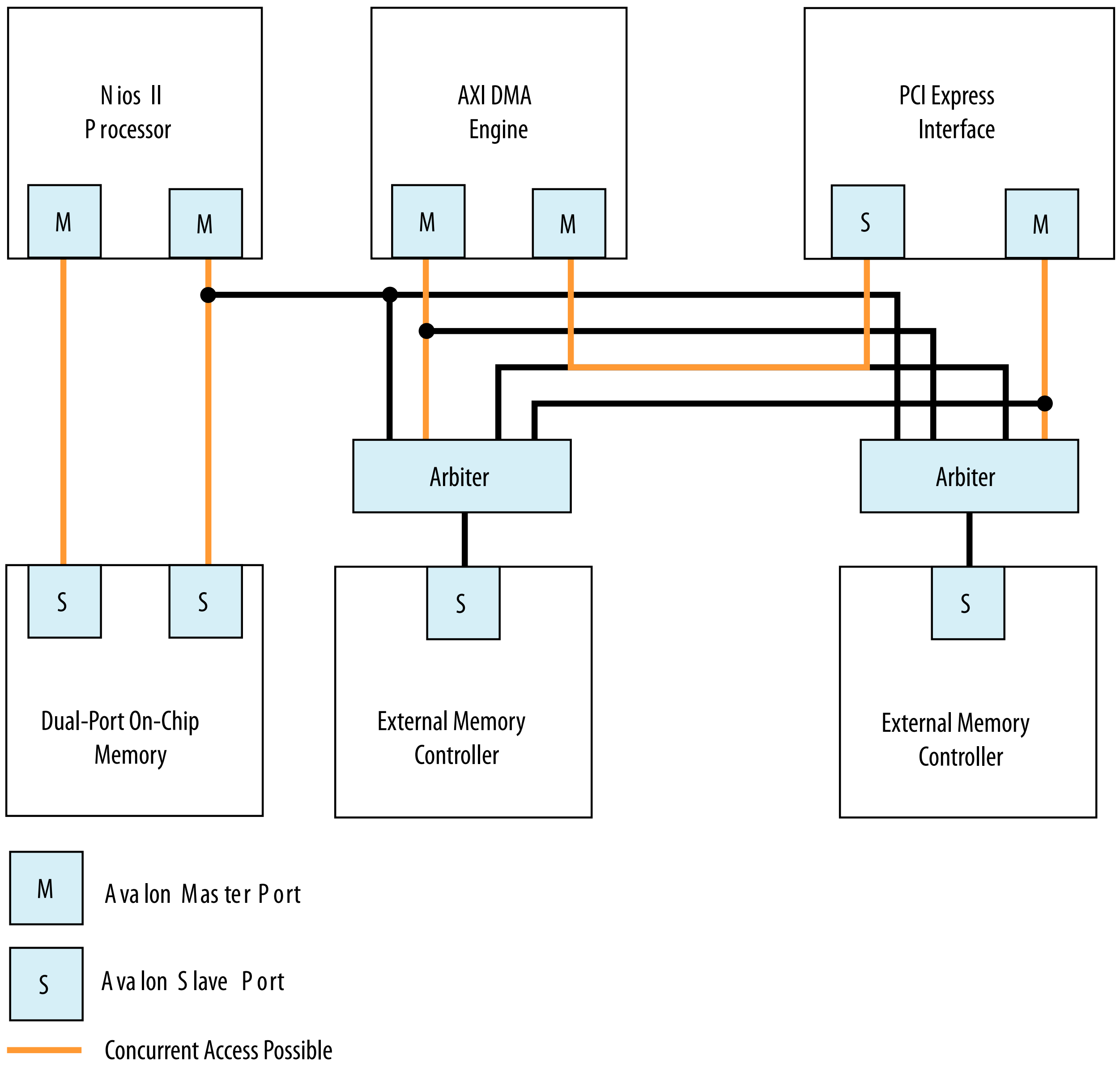
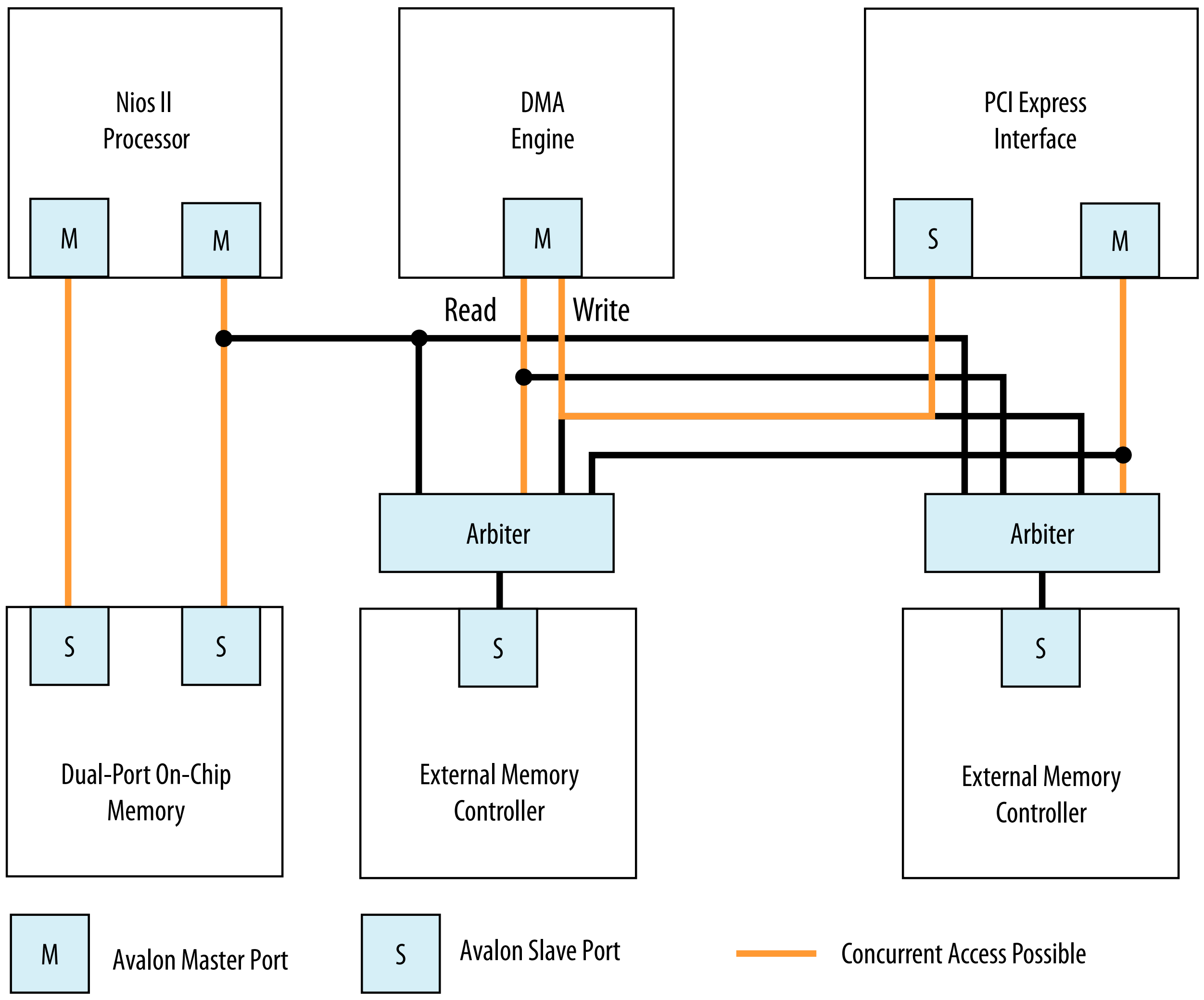


Figure 8-5: AXI Multiple Master Parallel Access

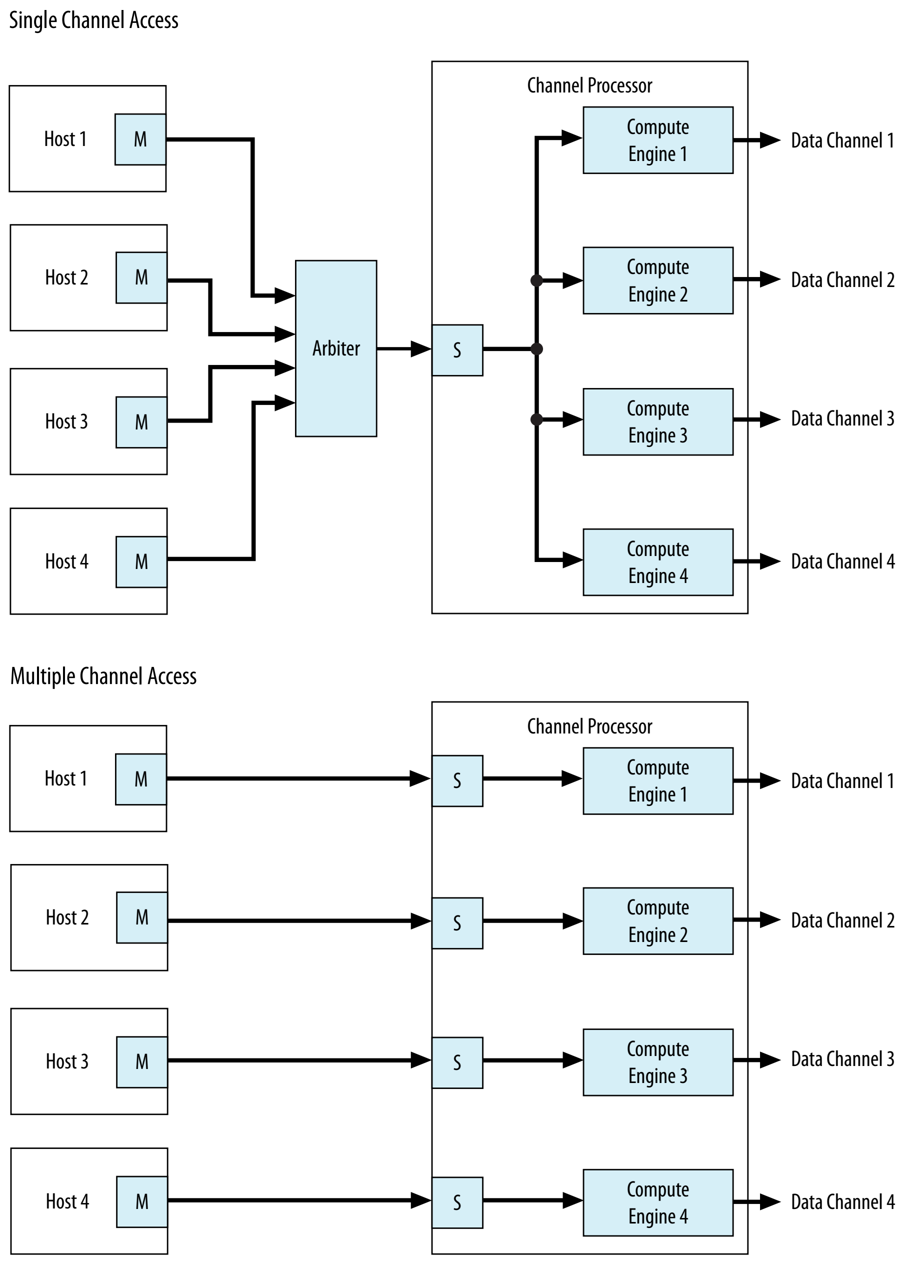
In this example, the DMA engine operates with a single master, because in AXI, the write and read channels on the master are independent and can process transactions simultaneously. There is concurrency between the read and write channels, with the yellow lines representing concurrent data paths.



Implementing Concurrency With Multiple Slaves

You can create multiple slave interfaces for a particular function to increase concurrency in your design.

Figure 8-6: Single Interface Versus Multiple Interfaces

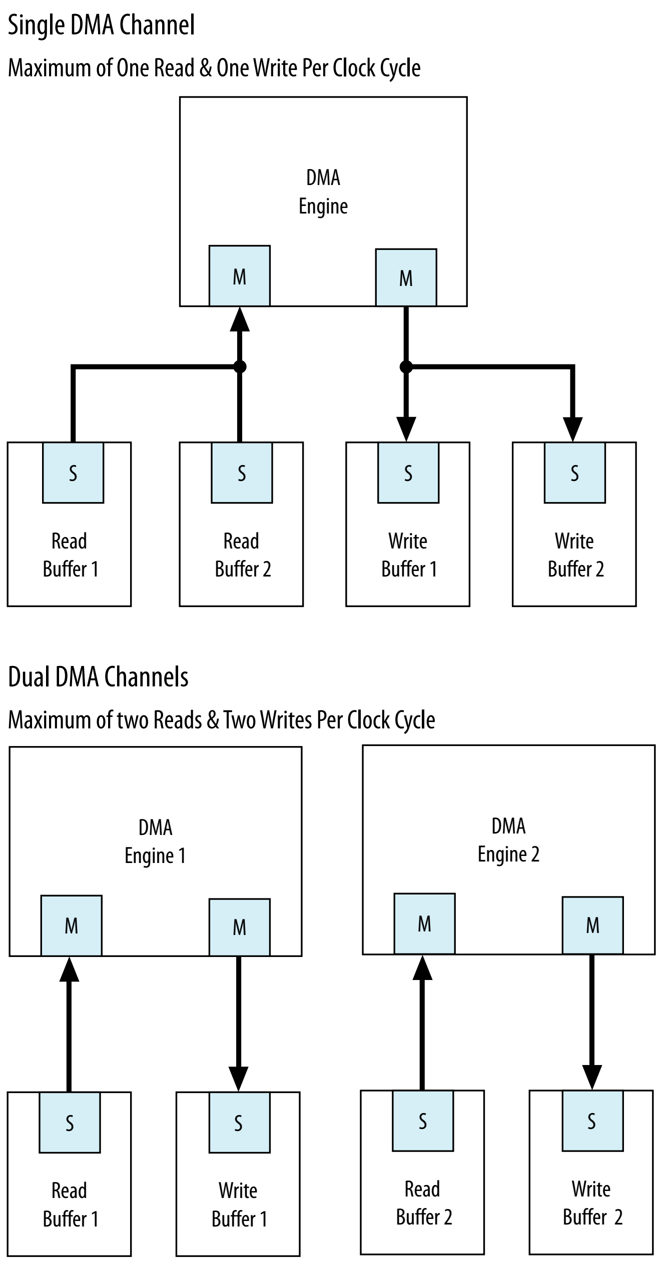


In this example, there are two channel processing systems. In the first, four hosts must arbitrate for the single slave interface of the channel processor. In the second, each host drives a dedicated slave interface, allowing all master interfaces to simultaneously access the slave interfaces of the component. Arbitration is not necessary when there is a single host and slave interface.

Implementing Concurrency with DMA Engines

In some systems, you can use DMA engines to increase throughput. You can use a DMA engine to transfer blocks of data between interfaces, which then frees the CPU from doing this task. A DMA engine transfers data between a programmed start and end address without intervention, and the data throughput is dictated by the components connected to the DMA. Factors that affect data throughput include data width and clock frequency.

Figure 8-7: Single or Dual DMA Channels



In this example, the system can sustain more concurrent read and write operations by including more DMA engines. Accesses to the read and write buffers in the top system are split between two DMA engines, as shown in the Dual DMA Channels at the bottom of the figure.

The DMA engine operates with Avalon-MM write and read masters. An AXI DMA typically has only one master, because in AXI, the write and read channels on the master are independent and can process transactions simultaneously.

Inserting Pipeline Stages to Increase System Frequency

Qsys provides the **Limit interconnect pipeline stages to** option on the **Project Settings** tab to automatically add pipeline stages to the Qsys interconnect when you generate a system.

You can specify between 0 to 4 pipeline stages, where 0 means that the interconnect has a combinational data path. You can specify a unique interconnect pipeline stage value for each subsystem.

Adding pipeline stages may increase the f_{MAX} of the design by reducing the combinational logic depth, at the cost of additional latency and logic utilization.

The insertion of pipeline stages requires certain interconnect components. For example, in a system with a single slave interface, there is no multiplexer; therefore multiplexer pipelining does not occur. When there is an Avalon or AXI single-master to single-slave system, no pipelining occurs, regardless of the **Limit interconnect pipeline stages to** option.

Related Information

- **Creating a System with Qsys** on page 5-1

Using Bridges

You can use bridges to increase system frequency, minimize generated Qsys logic, minimize adapter logic, and to structure system topology when you want to control where Qsys adds pipelining. You can also use bridges with arbiters when there is concurrency in the system.

An Avalon bridge has an Avalon-MM slave interface and an Avalon-MM master interface. You can have many components connected to the bridge slave interface, or many components connected to the bridge master interface. You can also have a single component connected to a single bridge slave or master interface.

You can configure the data width of the bridge, which can affect how Qsys generates bus sizing logic in the interconnect. Both interfaces support Avalon-MM pipelined transfers with variable latency, and can also support configurable burst lengths.

Transfers to the bridge slave interface are propagated to the master interface, which connects to components downstream from the bridge. When you need greater control over interconnect pipelining, you can use bridges instead of the **Limit Interconnect Pipeline Stages to** option.

Note: You can use Avalon bridges between AXI interfaces, and between Avalon domains. Qsys automatically creates interconnect logic between the AXI and Avalon interfaces, so you do not have to explicitly instantiate bridges between these domains. For more discussion about the benefits and disadvantages of shared and separate domains, refer to the *Qsys Interconnect*.

Related Information

- **Creating a System with Qsys** on page 5-1
- **Qsys Interconnect** on page 7-1

Using Bridges to Increase System Frequency

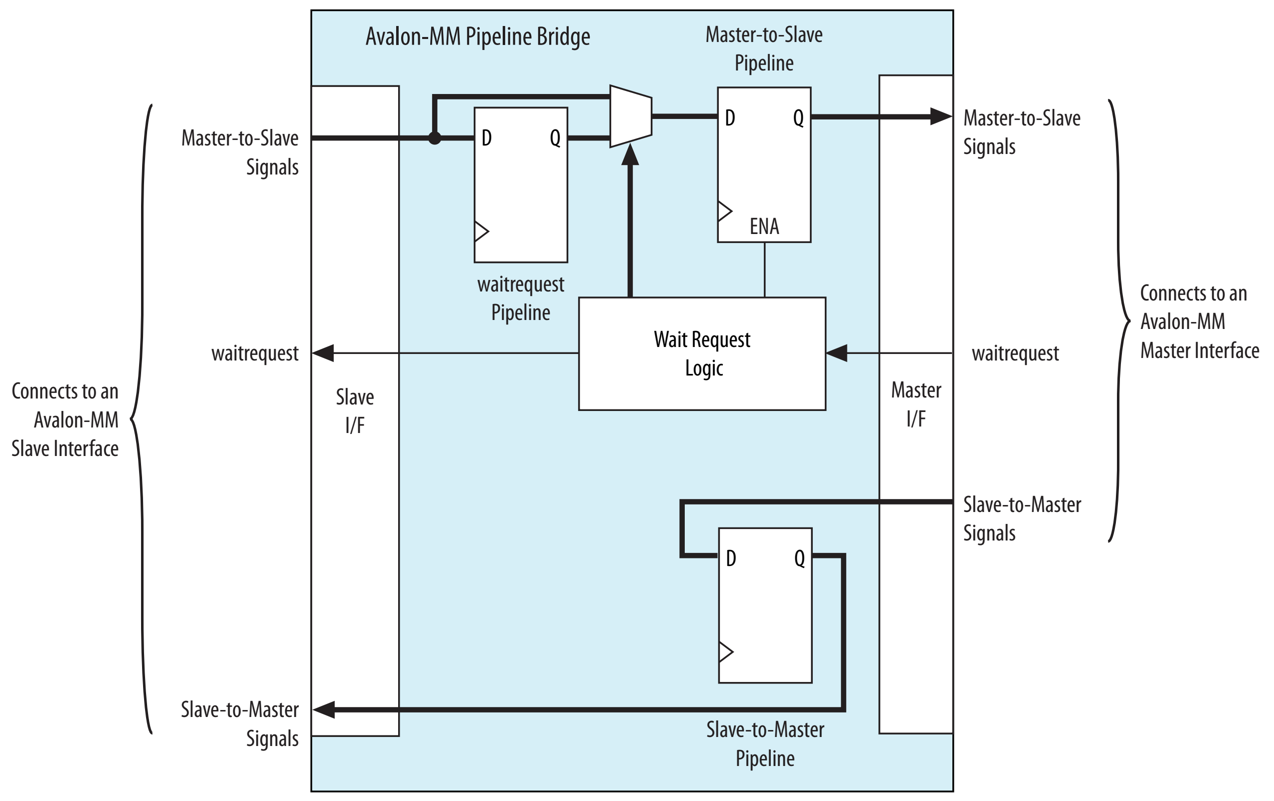
In Qsys, you can introduce interconnect pipeline stages or pipeline bridges to increase clock frequency in your system. Bridges control the system interconnect topology and allow you to subdivide the interconnect, giving you more control over pipelining and clock crossing functionality.

Inserting Pipeline Bridges

You can insert an Avalon-MM pipeline bridge to insert registers in the path between the bridges and its master and slaves. If a critical register-to-register delay occurs in the interconnect, a pipeline bridge can help reduce this delay and improve system f_{MAX}.

The Avalon-MM pipeline bridge component integrates into any Qsys system. The pipeline bridge options can increase logic utilization and read latency. The change in topology may also reduce concurrency if multiple masters arbitrate for the bridge. You can use the Avalon-MM pipeline bridge to control topology without adding a pipeline stage. A pipeline bridge that does not add a pipeline stage is optimal in some latency-sensitive applications. For example, a CPU may benefit from minimal latency when accessing memory.

Figure 8-8: Avalon-MM Pipeline Bridge

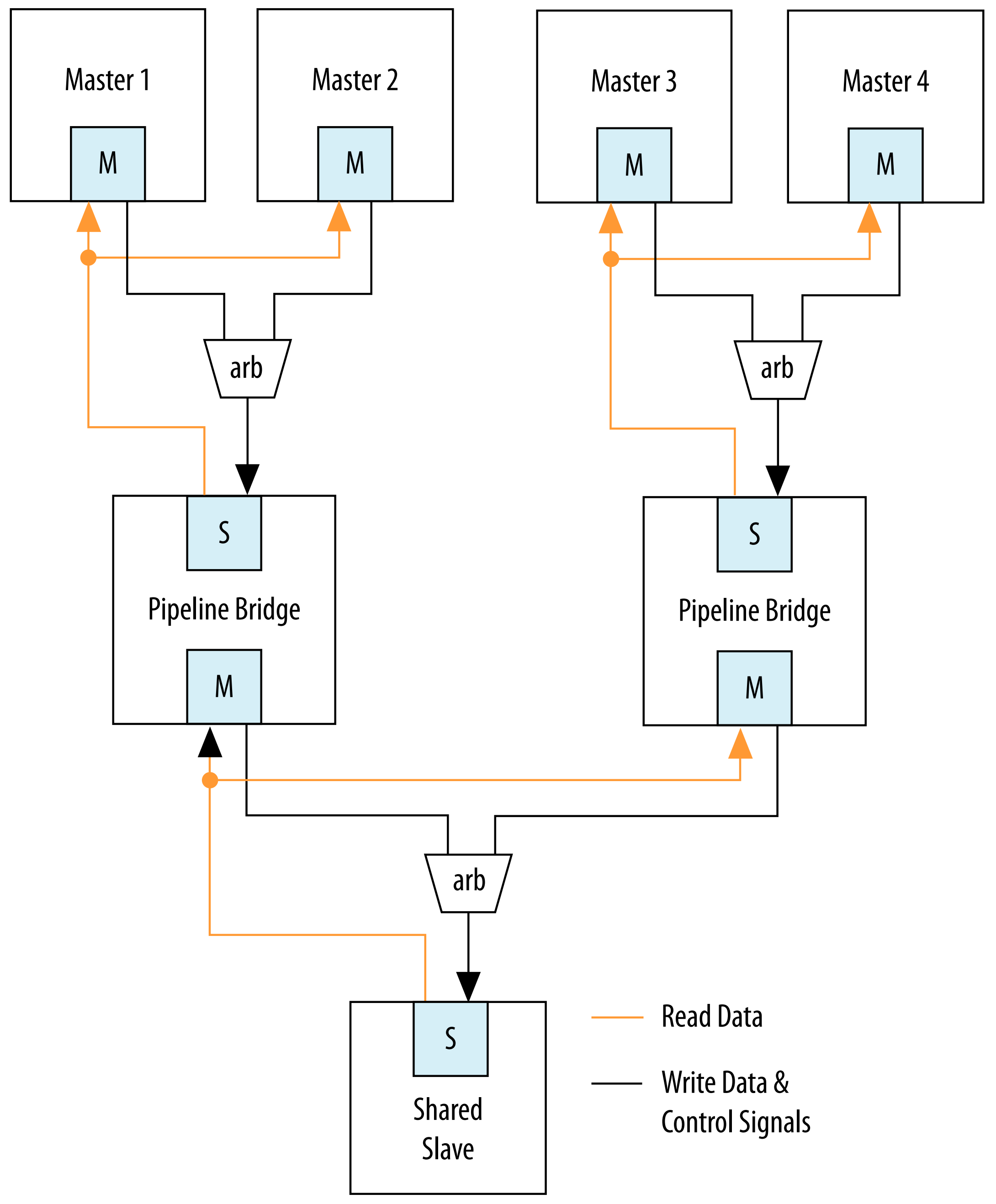


Implementing Command Pipelining (Master-to-Slave)

When multiple masters share a slave device, you can use command pipelining to improve performance.

The arbitration logic for the slave interface must multiplex the `address`, `writedata`, and `burstcount` signals. The multiplexer width increases proportionally with the number of masters connecting to a single slave interface. The increased multiplexer width may become a timing critical path in the system. If a single pipeline bridge does not provide enough pipelining, you can instantiate multiple instances of the bridge in a tree structure to increase the pipelining and further reduce the width of the multiplexer at the slave interface.

Figure 8-9: Tree of Bridges



Implementing Response Pipelining (Slave-to-Master)

When masters connect to multiple slaves that support read transfers, you can use slave-to-master pipelining to improve performance.

The interconnect inserts a multiplexer for every read data path back to the master. As the number of slaves supporting read transfers connecting to the master increases, the width of the read data multiplexer also increases. If the performance increase is insufficient with one bridge, you can use multiple bridges in a tree structure to improve f_{MAX}.

Using Clock Crossing Bridges

The clock crossing bridge contains a pair of clock crossing FIFOs, which isolate the master and slave interfaces in separate, asynchronous clock domains. Transfers to the slave interface are propagated to the master interface.

When you use a FIFO clock crossing bridge for the clock domain crossing, you add data buffering. Buffering allows pipelined read masters to post multiple reads to the bridge, even if the slaves downstream from the bridge do not support pipelined transfers.

You can also use a clock crossing bridge to place high and low frequency components in separate clock domains. If you limit the fast clock domain to the portion of your design that requires high performance, you may achieve a higher f_{MAX} for this portion of the design. For example, the majority of processor peripherals in embedded designs do not need to operate at high frequencies, therefore, you do not need to use a high-frequency clock for these components. When you compile a design with the Quartus II software, compilation may take more time when the clock frequency requirements are difficult to meet because the Fitter needs more time to place registers to achieve the required f_{MAX}. To reduce the amount of effort that the Fitter uses on low priority and low performance components, you can place these behind a clock crossing bridge operating at a lower frequency, allowing the Fitter to increase the effort placed on the higher priority and higher frequency data paths.

Using Bridges to Minimize Design Logic

Bridges can reduce interconnect logic by reducing the amount of arbitration and multiplexer logic that Qsys generates. This reduction occurs because bridges limit the number of concurrent transfers that can occur.

Avoiding Speed Optimizations That Increase Logic

You can add an additional pipeline stage with a pipeline bridge between masters and slaves to reduces the amount of combinational logic between registers, which can increase system performance. If you can increase the f_{MAX} of your design logic, you may be able to turn off the Quartus II software optimization settings, such as the **Perform register duplication** setting. Register duplication creates duplicate registers in two or more physical locations in the FPGA to reduce register-to-register delays. You may also want to choose **Speed** for the optimization method, which typically results in higher logic utilization due to logic duplication. By making use of the registers or FIFOs available in the bridges, you can increase the design speed and avoid needless logic duplication or speed optimizations, thereby reducing the logic utilization of the design.

Limiting Concurrency

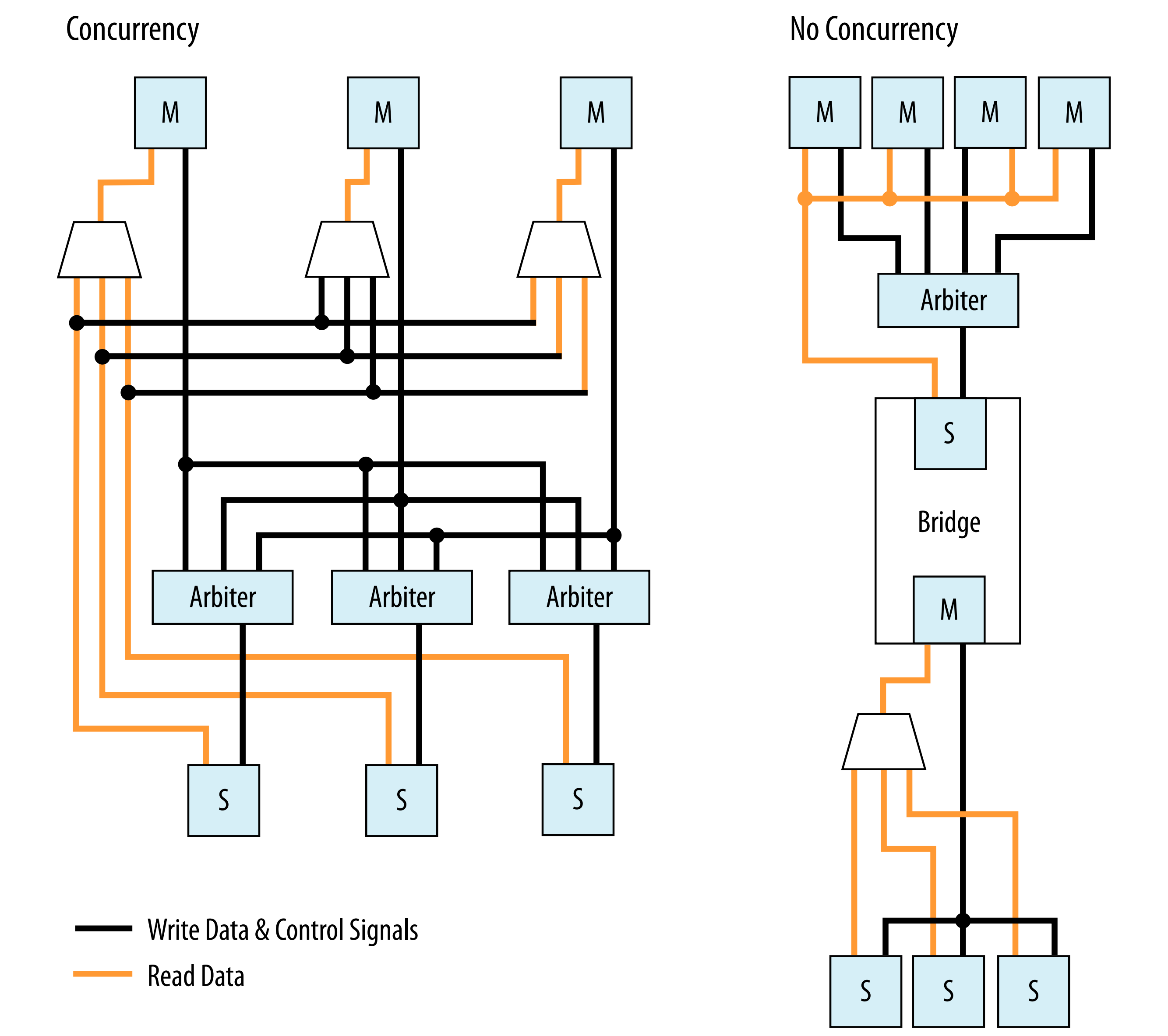
The amount of logic generated for the interconnect often increases as the system becomes larger because Qsys creates arbitration logic for every slave interface that is shared by multiple master interfaces. Qsys inserts multiplexer logic between master interfaces that connect to multiple slave interfaces if both support read data paths.

Most embedded processor designs contain components that are either incapable of supporting high data throughput, or do not need to be accessed frequently. These components can contain master or slave interfaces. Because the interconnect supports concurrent accesses, you may want to limit concurrency by inserting bridges into the data path to limit the amount of arbitration and multiplexer logic generated.

For example, if a system contains three master and three slave interfaces that are interconnected, Qsys generates three arbiters and three multiplexers for the read data path. If these masters do not require a significant amount of simultaneous throughput, you can reduce the resources that your design consumes by connecting the three masters to a pipeline bridge. The bridge controls the three slave interfaces and reduces the interconnect into a bus structure. Qsys creates one arbitration block between the bridge and the three masters, and a single read data path multiplexer between the bridge and three slaves, and prevents concurrency. This implementation is similar to a standard bus architecture.

You should not use this method for high throughput data paths to ensure that you do not limit overall system performance.

Figure 8-10: Differences Between Systems With and Without a Pipeline Bridge



Using Bridges to Minimize Adapter Logic

Qsys generates adapter logic for clock crossing, width adaptation, and burst support when there is a mismatch between the clock domains, widths, or bursting capabilities of the master and slave interface pairs.

Qsys creates burst adapters when the maximum burst length of the master is greater than the master burst length of the slave. The adapter logic creates extra logic resources, which can be substantial when your system contains master interfaces connected to many components that do not share the same characteristics. By placing bridges in your design, you can reduce the amount of adapter logic that Qsys generates.

Determining Effective Placement of Bridges

To determine the effective placement of a bridge, you should initially analyze each master in your system to determine if the connected slave devices support different bursting capabilities or operate in a different clock domain. The maximum burstcount of a component is visible as the `burstcount` signal in the HDL file of the component. The maximum burst length is $2^{(\text{width(burstcount -1)})}$, therefore, if the `burstcount` width is four bits, the maximum `burstcount` is eight. If no `burstcount` signal is present, the component does not support bursting or has a burst length of 1.

To determine if the system requires a clock crossing adapter between the master and slave interfaces, check the **Clock** column for the master and slave interfaces. If the clock is different for the master and slave interfaces, Qsys inserts a clock crossing adapter between them. To avoid creating multiple adapters, you can place the components containing slave interfaces behind a bridge so that Qsys creates a single adapter. By placing multiple components with the same burst or clock characteristics behind a bridge, you limit concurrency and the number of adapters.

You can also use a bridge to separate AXI and Avalon domains to minimize burst adaptation logic. For example, if there are multiple Avalon slaves that are connected to an AXI master, you can consider inserting a bridge to access the adaptation logic once before the bridge, instead of once per slave. This implementation results in latency, and you would also lose concurrency between reads and writes.

Changing the Response Buffer Depth

When you use automatic clock-crossing adapters, Qsys determines the required depth of FIFO buffering based on the slave properties. If a slave has a high *Maximum Pending Reads* parameter, the resulting deep response buffer FIFO that Qsys inserts between the master and slave can consume a lot of device resources. To control the response FIFO depth, you can use a clock crossing bridge and manually adjust its FIFO depth to trade off throughput with smaller memory utilization.

For example, if you have masters that cannot saturate the slave, you do not need response buffering. Using a bridge reduces the FIFO memory depth and reduces the **Maximum Pending Reads** available from the slave.

Considering the Effects of Using Bridges

Before you use pipeline or clock crossing bridges in a design, you should carefully consider their effects. Bridges can have any combination of consequences on your design, which could be positive or negative. Benchmarking your system before and after inserting bridges can help you to determine the impact to the design.

Increased Latency

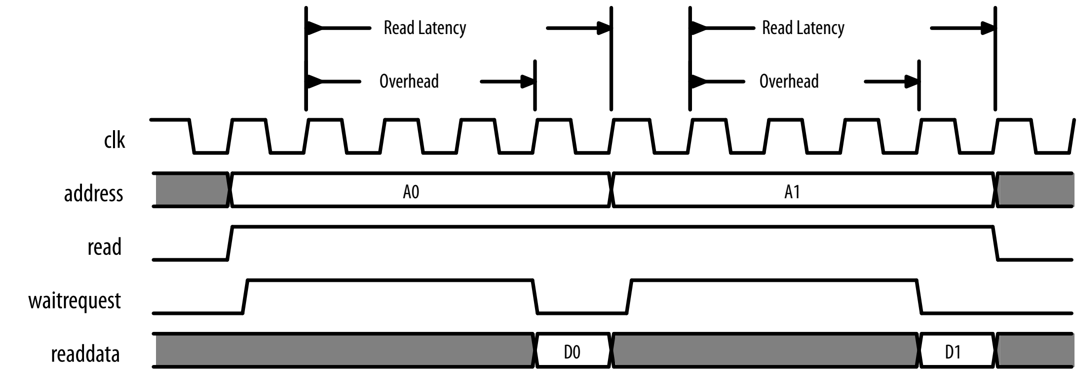
Adding a bridge to a design has an effect on the read latency between the master and the slave. Depending on the system requirements and the type of master and slave, this latency increase may or may not be acceptable in your design.

Acceptable Latency Increase

For a pipeline bridge, Qsys adds a cycle of latency for each pipeline option that is enabled. The buffering in the clock crossing bridge also adds latency. If you use a pipelined or burst master that posts many read transfers, the increase in latency does not impact performance significantly because the latency increase is very small compared to the length of the data transfer.

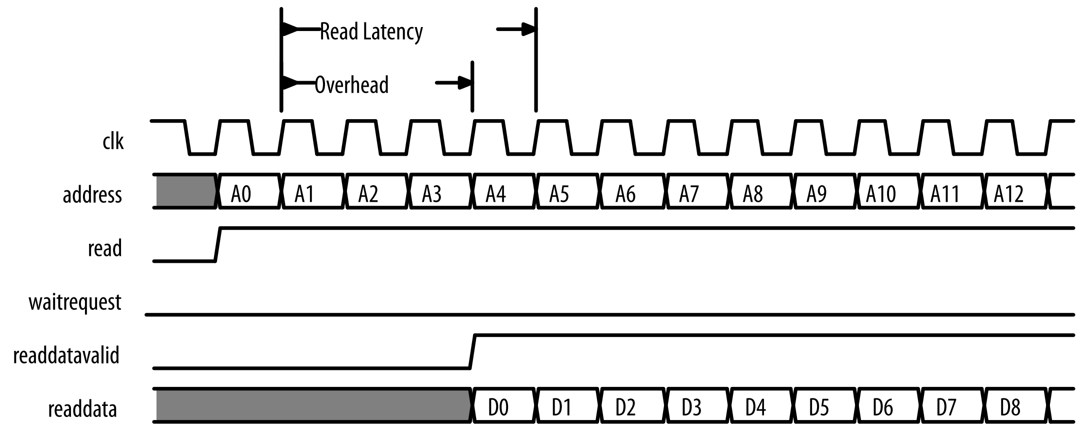
For example, if you use a pipelined read master such as a DMA controller to read data from a component with a fixed read latency of four clock cycles, but only perform a single word transfer, the overhead is three clock cycles out of the total of four. This is true when there is no additional pipeline latency in the interconnect. The read throughput is only 25%.

Figure 8-11: Low-Efficiency Read Transfer



However, if 100 words of data are transferred without interruptions, the overhead is three cycles out of the total of 103 clock cycles. This corresponds to a read efficiency of approximately 97% when there is no additional pipeline latency in the interconnect. Adding a pipeline bridge to this read path adds two extra clock cycles of latency. The transfer requires 105 cycles to complete, corresponding to an efficiency of approximately 94%. Although the efficiency decreased by 3%, adding the bridge may increase the f_{MAX} by 5%. For example, if the clock frequency can be increased, the overall throughput would improve. As the number of words transferred increases, the efficiency increases to nearly 100%, whether or not a pipeline bridge is present.

Figure 8-12: High Efficiency Read Transfer

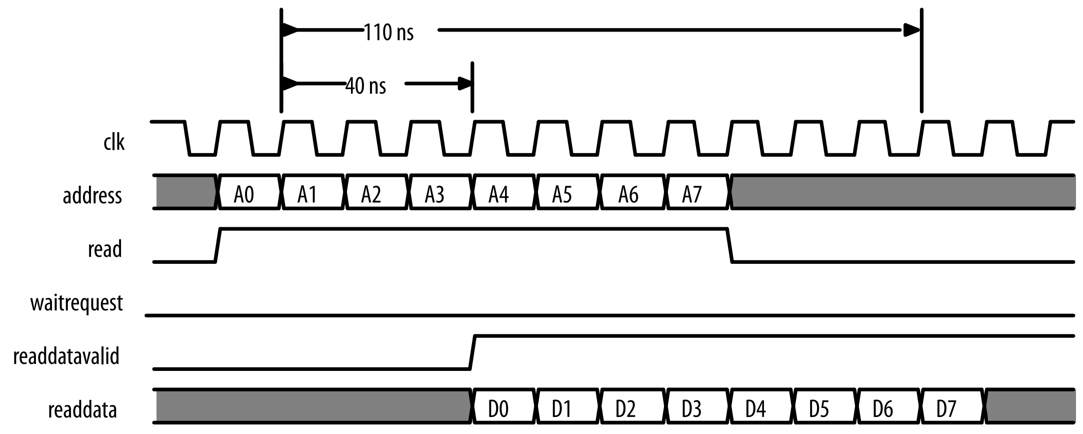


Unacceptable Latency Increase

Processors are sensitive to high latency read times and typically retrieve data for use in calculations that cannot proceed until the data arrives. Before adding a bridge to the data path of a processor instruction or data master, determine whether the clock frequency increase justifies the added latency.

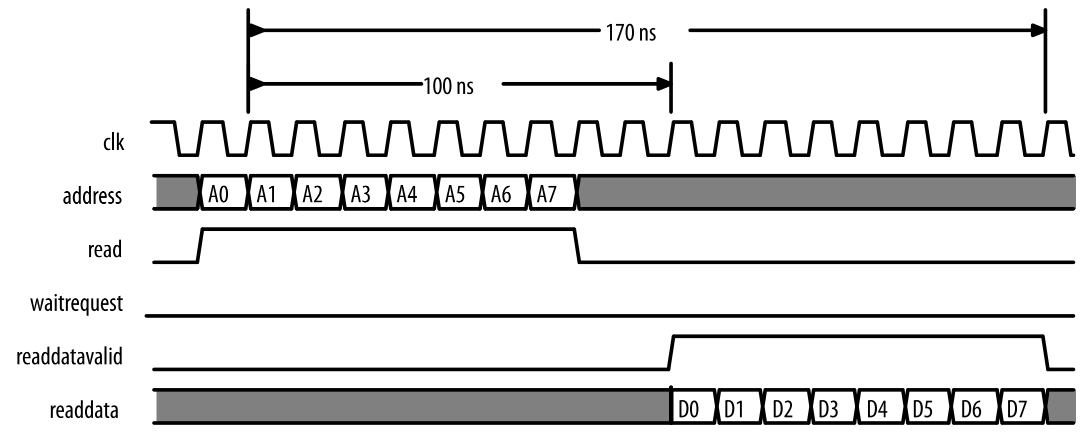
A Nios II processor instruction master has a cache memory with a read latency of four cycles, which is eight sequential words of data return for each read. At 100 MHz, the first read takes 40 ns to complete. Each successive word takes 10 ns so that eight reads complete in 110 ns.

Figure 8-13: Performance of a Nios II Processor and Memory Operating at 100 MHz



Adding a clock crossing bridge allows the memory to operate at 125 MHz. However, this increase in frequency is negated by the increase in latency because if the clock crossing bridge adds six clock cycles of latency at 100 MHz, then the memory continues to operate with a read latency of four clock cycles. Consequently, the first read from memory takes 100 ns, and each successive word takes 10 ns because reads arrive at the frequency of the processor, which is 100 MHz. In total, eight reads complete after 170 ns. Although the memory operates at a higher clock frequency, the frequency at which the master operates limits the throughput.

Figure 8-14: Performance of a Nios II Processor and Eight Reads with Ten Cycles Latency

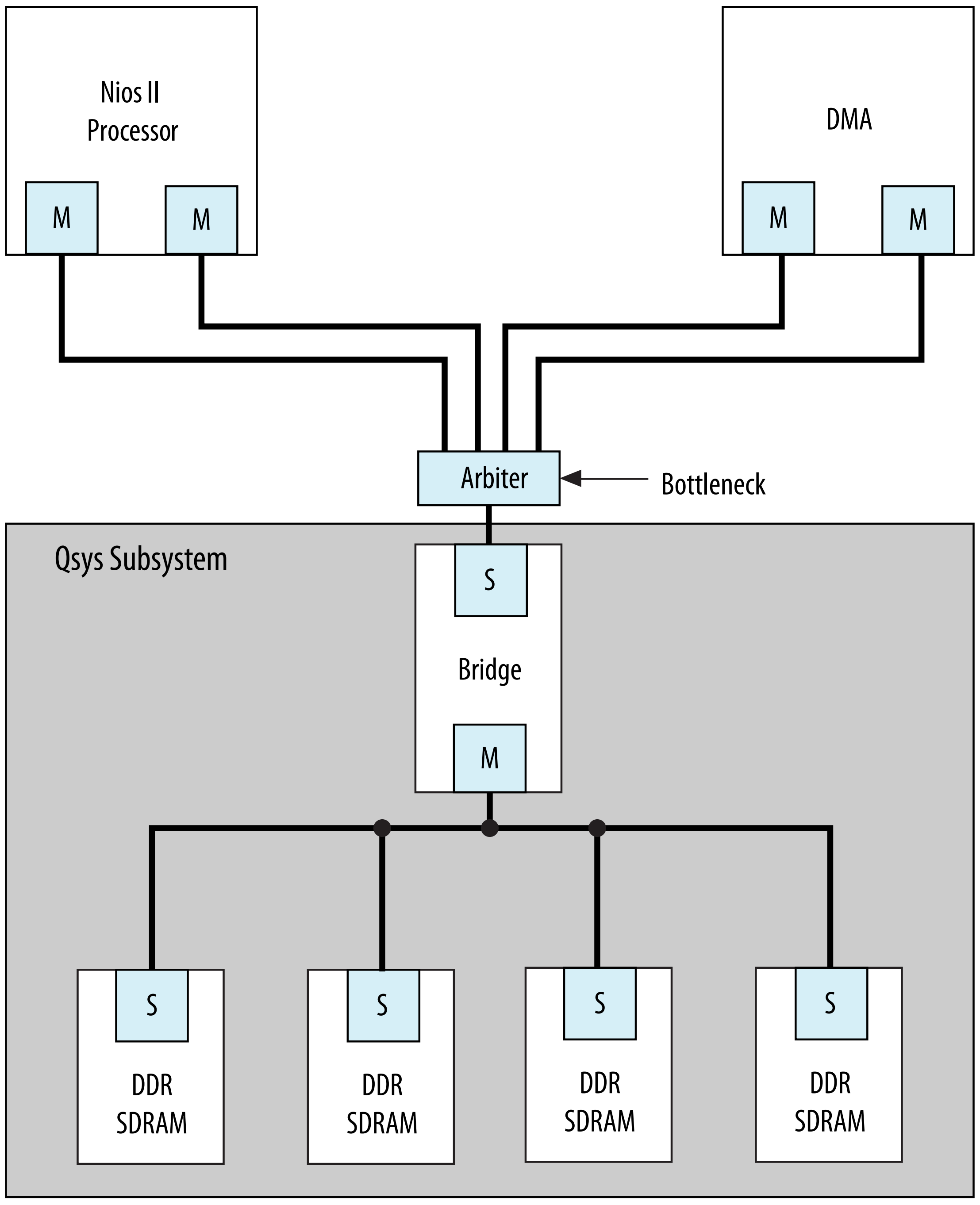


Limited Concurrency

Placing a bridge between multiple master and slave interfaces limits the number of concurrent transfers your system can initiate. This limitation is the same when connecting multiple master interfaces to a single slave interface. The slave interface of the bridge is shared by all the masters and, as a result, Qsys creates arbitration logic. If the components placed behind a bridge are infrequently accessed, this concurrency limitation may be acceptable.

Bridges can have a negative impact on system performance if you use them inappropriately. For example, if multiple memories are used by several masters, you should not place the memory components behind a bridge. The bridge limits memory performance by preventing concurrent memory accesses. Placing multiple memory components behind a bridge can cause the separate slave interfaces to appear as one large memory to the masters accessing the bridge; all masters must access the same slave interface.

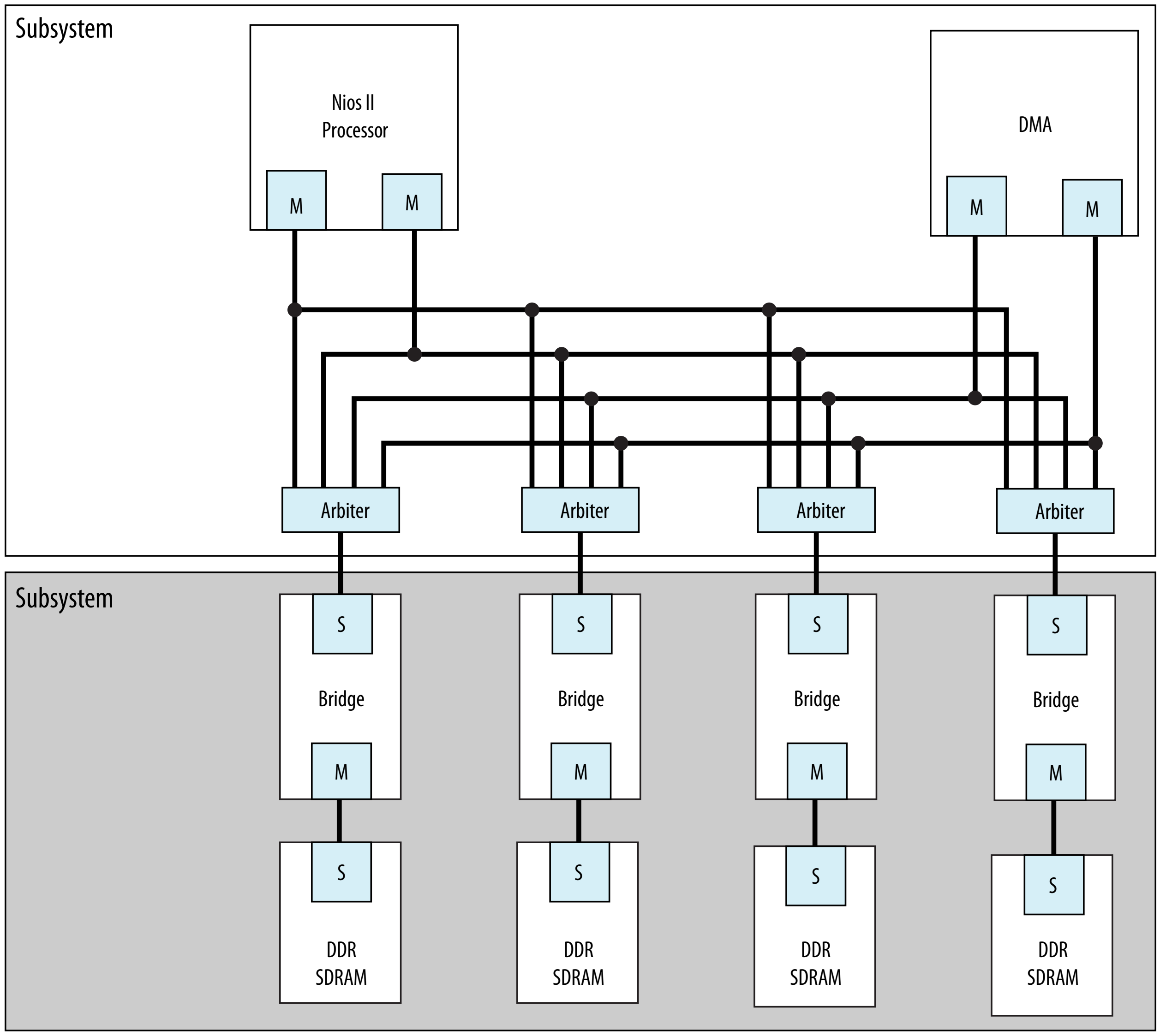
Figure 8-15: Inappropriate Use of a Bridge in a Hierarchical System



A memory subsystem with one bridge that acts as a single slave interface for the Avalon-MM Nios II and DMA masters, which results in a bottleneck architecture. The bridge acts as a bottleneck between the two masters and the memories.

If the f_{MAX} of your memory interfaces is low and you want to use a pipeline bridge between subsystems, you can place each memory behind its own bridge, which increases the f_{MAX} of the system without sacrificing concurrency.

Figure 8-16: Efficient Memory Pipelining Without a Bottleneck in a Hierarchical System

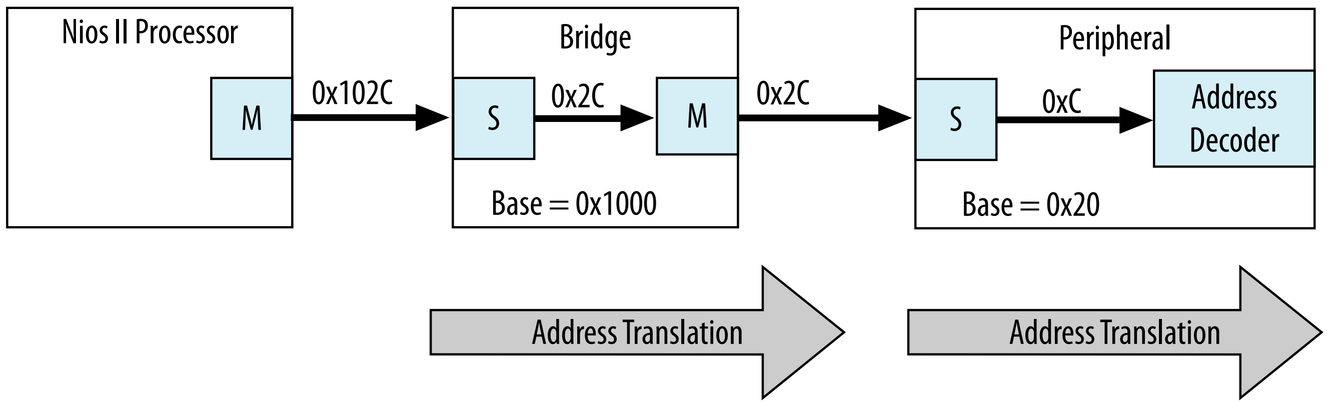


Address Space Translation

The slave interface of a pipeline or clock crossing bridge has a base address and address span. You can set the base address, or allow Qsys to set it automatically. The address of the slave interface is the base offset address of all the components connected to the bridge. The address of components connected to the bridge is the sum of the base offset and the address of that component.

The master interface of the bridge drives only the address bits that represent the offset from the base address of the bridge slave interface. Any time a master accesses a slave through a bridge, both addresses must be added together, otherwise the transfer fails. The **Address Map** tab displays the addresses of the slaves connected to each master and includes address translations caused by system bridges.

Figure 8-17: Bridge Address Translation

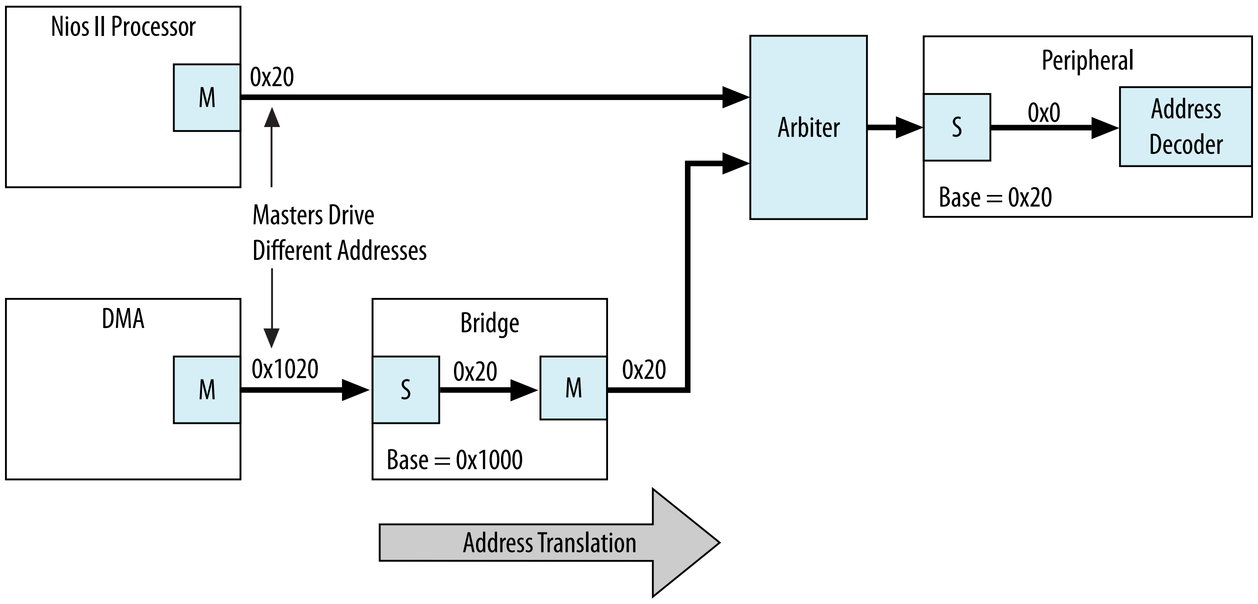


In this example, the Nios II processor connects to a bridge located at base address 0x1000, a slave connects to the bridge master interface at an offset of 0x20, and the processor performs a write transfer to the fourth 32-bit or 64-bit word within the slave. Nios II drives the address 0x102C to interconnect, which is within the address range of the bridge. The bridge master interface drives 0x2C, which is within the address range of the slave, and the transfer completes.

Address Coherency

To simplify the system design, all masters should access slaves at the same location. In many systems, a processor passes buffer locations to other mastering components, such as a DMA controller. If the processor and DMA controller do not access the slave at the same location, Qsys must compensate for the differences.

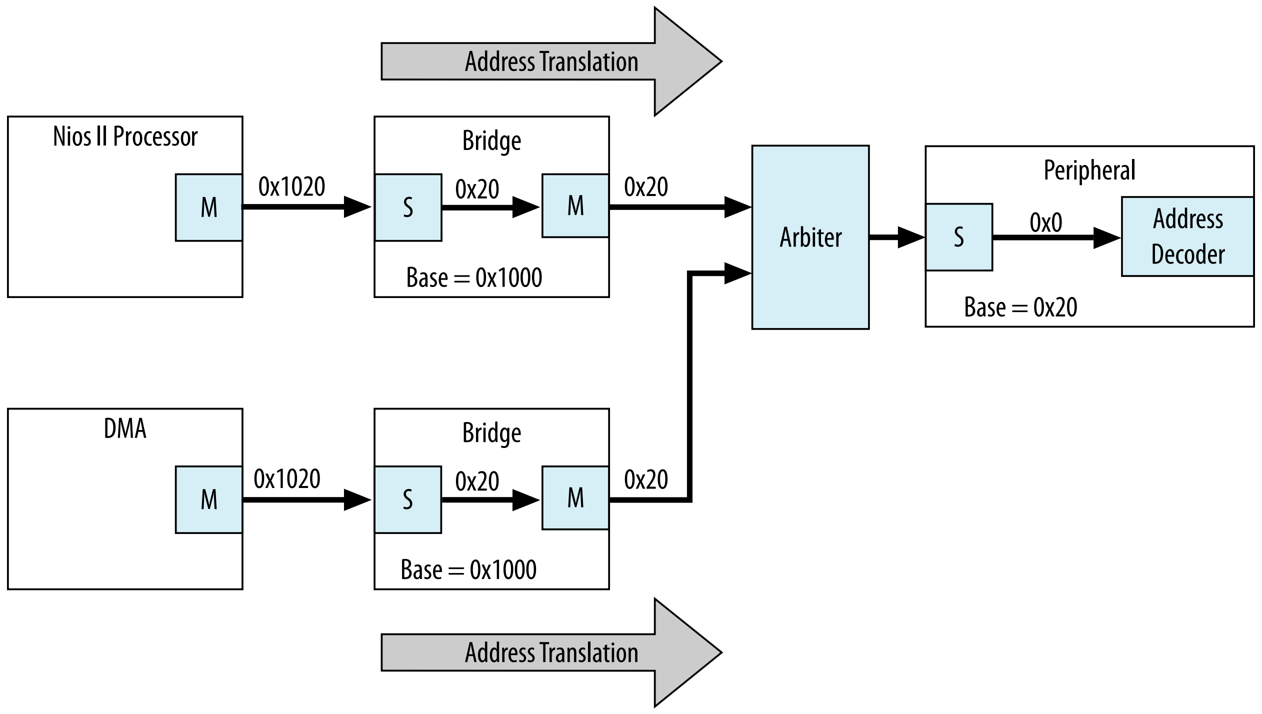
Figure 8-18: Slaves at Different Addresses and Complicating the System



A Nios II processor and DMA controller access a slave interface located at address 0x20. The processor connects directly to the slave interface. The DMA controller connects to a pipeline bridge located at address 0x1000, which then connects to the slave interface. Because the DMA controller accesses the pipeline bridge first, it must drive 0x1020 to access the first location of the slave interface. Because the processor accesses the slave from a different location, you must maintain two base addresses for the slave device.

To avoid the requirement for two addresses, you can add an additional bridge to the system, set its base address to 0x1000, and then disable all the pipelining options in the second bridge so that the bridge has minimal impact on system timing and resource utilization. Because this second bridge has the same base address as the original bridge, the processor and DMA controller access the slave interface with the same address range.

Figure 8-19: Address Translation Corrected With Bridge



Increasing Transfer Throughput

Increasing the transfer efficiency of the master and slave interfaces in your system increases the throughput of your design. Designs with strict cost or power requirements benefit from increasing the transfer efficiency because you can then use less expensive, lower frequency devices. Designs requiring high performance also benefit from increased transfer efficiency because increased efficiency improves the performance of frequency–limited hardware.

Throughput is the number of symbols (such as bytes) of data that Qsys can transfer in a given clock cycle. Read latency is the number of clock cycles between the address and data phase of a transaction. For example, a read latency of two means that the data is valid two cycles after the address is posted. If the master must wait for one request to finish before the next begins, such as with a processor, then the read latency is very important to the overall throughput.

You can measure throughput and latency in simulation by observing the waveforms, or using the verification IP monitors.

Related Information

- **Avalon Verification IP Suite User Guide**
- **Mentor® Verification IP Altera Edition AMBA AXI3/4 User Guide**

Using Pipelined Transfers

Pipelined transfers increase the read efficiency by allowing a master to post multiple reads before data from an earlier read returns. Masters that support pipelined transfers post transfers continuously, relying on the `readdatavalid` signal to indicate valid data. Slaves support pipelined transfers by including the `readdatavalid` signal or operating with a fixed read latency.

AXI masters declare how many outstanding writes and reads it can issue with the `writeIssuingCapability` and `readIssuingCapability` parameters. In the same way, a slave can declare how many reads it can accept with the `readAcceptanceCapability` parameter. AXI masters with a read issuing capability greater than one are pipelined in the same way as Avalon masters and the `readdatavalid` signal.

Using the Maximum Pending Reads Parameter

If you create a custom component with a slave interface supporting variable-latency reads, you must specify the **Maximum Pending Reads** parameter in the Component Editor. Qsys uses this parameter to generate the appropriate interconnect and represent the maximum number of read transfers that your pipelined slave component can process. If the number of reads presented to the slave interface exceeds the **Maximum Pending Reads** parameter, then the slave interface must assert `waitrequest`.

Optimizing the value of the **Maximum Pending Reads** parameter requires an understanding of the latencies of your custom components. This parameter should be based on the component's highest read latency for the various logic paths inside the component. For example, if your pipelined component has two modes, one requiring two clock cycles and the other five, set the **Maximum Pending Reads** parameter to 5 to allow your component to pipeline five transfers, and eliminating dead cycles after the initial five-cycle latency.

You can also determine the correct value for the **Maximum Pending Reads** parameter by monitoring the number of reads that are pending during system simulation or while running the hardware. To use this method, set the parameter to a high value and use a master that issues read requests on every clock. You can use a DMA for this task as long as the data is written to a location that does not frequently assert `waitrequest`. If you implement this method, you can observe your component with a logic analyzer or built-in monitoring hardware.

Choosing the correct value for the **Maximum Pending Reads** parameter of your custom pipelined read component is important. If you underestimate the parameter value, you may cause a master interface to stall with a `waitrequest` until the slave responds to an earlier read request and frees a FIFO position.

The **Maximum Pending Reads** parameter controls the depth of the response FIFO inserted into the interconnect for each master connected to the slave. This FIFO does not use significant hardware resources. Overestimating the **Maximum Pending Reads** parameter results in a slight increase in hardware utilization. For these reasons, if you are not sure of the optimal value, you should overestimate this value.

If your system includes a bridge, you must set the **Maximum Pending Reads** parameter on the bridge as well. To allow maximum throughput, this value should be equal to or greater than the **Maximum Pending Reads** value for the connected slave that has the highest value. You can limit the maximum pending reads of a slave and reduce the buffer depth by reducing the parameter value on the bridge if the high throughput is not required. If you do not know the **Maximum Pending Reads** value for all the slave components, you can monitor the number of reads that are pending during system simulation while running the hardware. To use this method, set the **Maximum Pending Reads** parameter to a high value and use a master that issues read requests on every clock, such as a DMA. Then, reduce the number of maximum pending reads of the bridge until the bridge reduces the performance of any masters accessing the bridge.

Arbitration Shares and Bursts

Arbitration shares provide control over the arbitration process. By default, the arbitration algorithm allocates evenly, with all masters receiving one share.

You can adjust the arbitration process by assigning a larger number of shares to the masters that need greater throughput. The larger the arbitration share, the more transfers are allocated to the master to access a slave. The masters gets uninterrupted access to the slave for its number of shares, as long as the master is reading or writing.

If a master cannot post a transfer and other masters are waiting to gain access to a particular slave, the arbiter grants another master access. This mechanism prevents a master from wasting arbitration cycles if it cannot post back-to-back transfers. A bursting transaction contains multiple beats (or words) of data, starting from a single address. Bursts allow a master to maintain access to a slave for more than a single word transfer. If a bursting master posts a write transfer with a burst length of eight, it is guaranteed arbitration for eight write cycles.

You can assign arbitration shares to an Avalon-MM bursting master and AXI masters (which are always considered a bursting master). Each share consists of one burst transaction (such as multi-cycle write), and allows a master to complete a number of bursts before arbitration switches to the next master.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specification

Differences Between Arbitration Shares and Bursts

The following three key characteristics distinguish arbitration shares and bursts:

- Arbitration Lock
- Sequential Addressing
- Burst Adapters

Arbitration Lock

When a master posts a burst transfer, the arbitration is locked for that master; consequently, the bursting master should be capable of sustaining transfers for the duration of the locked period. If, after the fourth write, the master deasserts the write signal (Avalon-MM write or AXI `wvalid`) for fifty cycles, all other masters continue to wait for access during this stalled period.

To avoid wasted bandwidth, your master designs should wait until a full burst transfer is ready before requesting access to a slave device. Alternatively, you can avoid wasted bandwidth by posting `burstcounts` equal to the amount of data that is ready. For example, if you create a custom bursting write master with a maximum `burstcount` of eight, but only three words of data are ready, you can present a `burstcount` of three. This strategy does not result in optimal use of the system band width if the slave is capable of handling a larger burst; however, this strategy prevents stalling and allows access for other masters in the system.

Sequential Addressing

An Avalon-MM burst transfer includes a base address and a `burstcount`, which represents the number of words of data that are transferred, starting from the base address and incrementing sequentially. Burst transfers are common for processors, DMAs, and buffer processing accelerators; however, sometimes a master must access non-sequential addresses. Consequently, a bursting master must set the `burstcount` to the number of sequential addresses, and then reset the `burstcount` for the next location.

The arbitration share algorithm has no restrictions on addresses; therefore, your custom master can update the address it presents to the interconnect for every read or write transaction.

Burst Adapters

Qsys allows you to create systems that mix bursting and non-bursting master and slave interfaces. This design strategy allows you to connect bursting master and slave interfaces that support different maximum burst lengths, with Qsys generating burst adapters when appropriate.

Qsys inserts a burst adapter whenever a master interface burst length exceeds the burst length of the slave interface, or if the master issues a burst type that the slave cannot support. For example, if you connect an AXI master to an Avalon slave, a burst adapter is inserted. Qsys assigns non-bursting masters and slave interfaces a burst length of one. The burst adapter divides long bursts into shorter bursts. As a result, the burst adapter adds logic to the address and `burstcount` paths between the master and slave interfaces.

Related Information

Qsys Interconnect on page 7-1

AMBA Protocol Specification

Choosing Avalon-MM Interface Types

To avoid inefficient Avalon-MM transfers, custom master or slave interfaces must use the appropriate simple, pipelined, or burst interfaces.

Simple Avalon-MM Interfaces

Simple interface transfers do not support pipelining or bursting for reads or writes; consequently, their performance is limited. Simple interfaces are appropriate for transfers between masters and infrequently used slave interfaces. In Qsys, the PIO, UART, and Timer include slave interfaces that use simple transfers.

Pipelined Avalon-MM Interfaces

Pipelined read transfers allow a pipelined master interface to start multiple read transfers in succession without waiting for prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave port may require one or more cycles of latency to return data for each transfer.

In many systems, read throughput becomes inadequate if simple reads are used and pipelined transfers can increase throughput. If you define a component with a fixed read latency, Qsys automatically provides the pipelining logic necessary to support pipelined reads. You can use fixed latency pipelining as the default design starting point for slave interfaces. If your slave interface has a variable latency response time, use the `readdatavalid` signal to indicate when valid data is available. The interconnect implements read response FIFO buffering to handle the maximum number of pending read requests.

To use components that support pipelined read transfers, and to use a pipelined system interconnect efficiently, your system must contain pipelined masters. You can use pipelined masters as the default starting point for new master components. Use the `readdatavalid` signal for these master interfaces.

Because master and slaves sometimes have mismatched pipeline latency, interconnect contains logic to reconcile the differences.

Table 8-1: Pipeline Latency in a Master-Slave Pair

Master	Slave	Pipeline Management Logic Structure
No pipeline	No Pipeline	Qsys interconnect does not instantiate logic to handle pipeline latency.
No pipeline	Pipelined with fixed or variable latency	Qsys interconnect forces the master to wait through any slave-side latency cycles. This master-slave pair gains no benefits from pipelining, because the master waits for each transfer to complete before beginning a new transfer. However, while the master is waiting, the slave can accept transfers from a different master.
Pipelined	No pipeline	Qsys interconnect carries out the transfer as if neither master nor slave were pipelined, causing the master to wait until the slave returns data. An example of a non-pipeline slave is an asynchronous off-chip interface.
Pipelined	Pipelined with fixed latency	Qsys interconnect allows the master to capture data at the exact clock cycle when data from the slave is valid, to enable maximum throughput. An example of a fixed latency slave is an on-chip memory.
Pipelined	Pipelined with variable latency	The slave asserts a signal when its `readdata` is valid, and the master captures the data. The master-slave pair can achieve maximum throughput if the slave has variable latency. Examples of variable latency slaves include SDRAM and FIFO memories.

Burst Avalon-MM Interfaces

Burst transfers are commonly used for latent memories such as SDRAM and off-chip communication interfaces, such as PCI Express. To use a burst-capable slave interface efficiently, you must connect to a bursting master. Components that require bursting to operate efficiently typically have an overhead penalty associated with short bursts or non-bursting transfers.

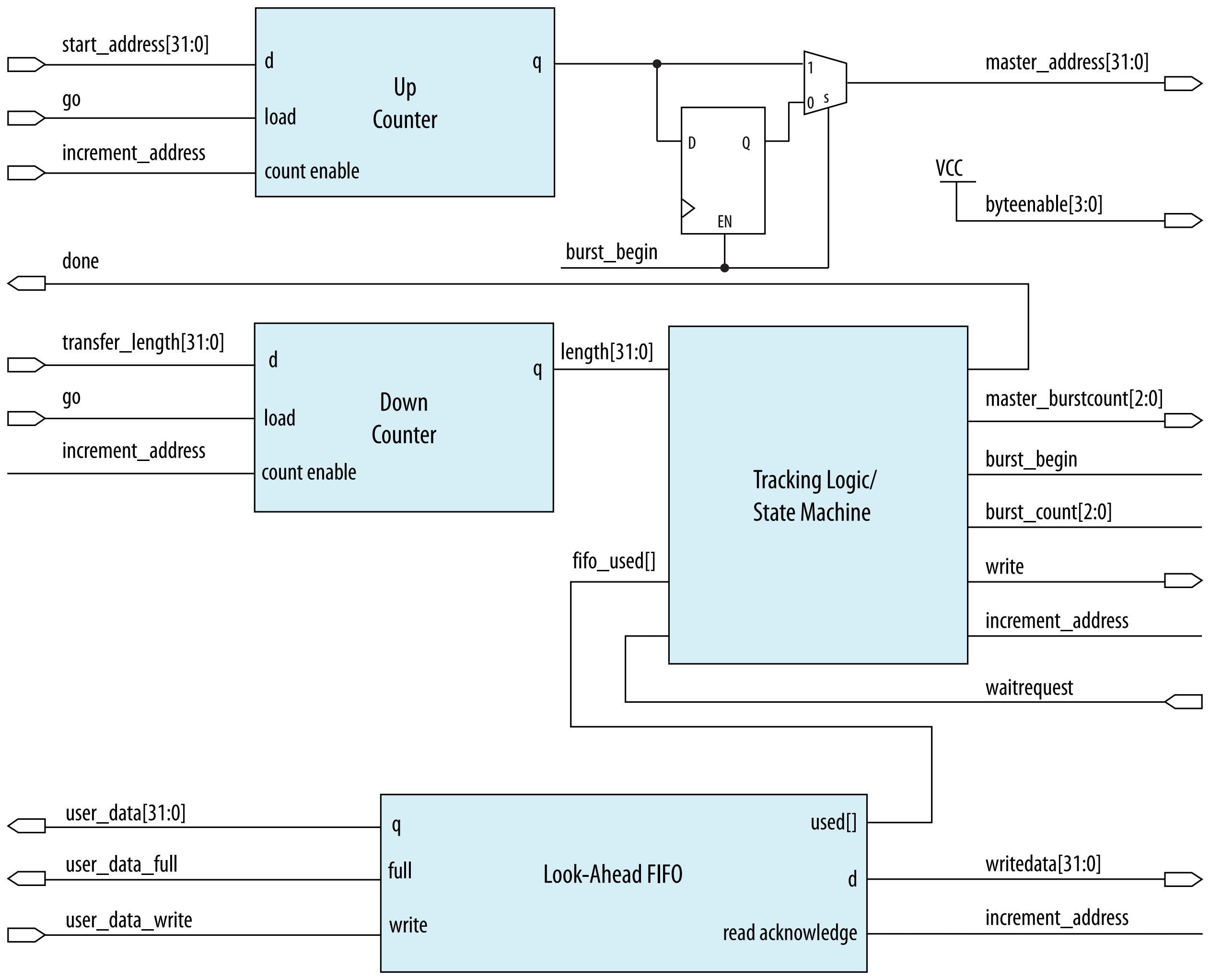
You can use a burst-capable slave interface if you know that your component requires sequential transfers to operate efficiently. Because SDRAM memories incur a penalty when switching banks or rows, performance improves when SDRAM memories are accessed sequentially with bursts.

Architectures that use the same signals to transfer address and data also benefit from bursting. Whenever an address is transferred over shared address and data signals, the throughput of the data transfer is reduced. Because the address phase adds overhead, using large bursts increases the throughput of the connection.

Avalon-MM Burst Master Example

Figure 8-20: Avalon Bursting Write Master

This example shows the architecture of a bursting write master that receives data from a FIFO and writes the contents to memory. You can use a bursting master as a starting point for your own bursting components, such as custom DMAs, hardware accelerators, or off-chip communication interfaces.



The master performs word accesses and writes to sequential memory locations. When go is asserted, the `start_address` and `transfer_length` are registered. On the next clock cycle, the control logic asserts `burst_begin`, which synchronizes the internal control signals in addition to the `master_address` and master_burstcount presented to the interconnect. The timing of these two signals is important because during bursting write transfers, byteenable, and burstcount must be held constant for the entire burst.

To avoid inefficient writes, the master posts a burst when enough data is buffered in the FIFO. To maximize the burst efficiency, the master should stall only when a slave asserts waitrequest. In this example, the FIFO's used signal tracks the number of words of data that are stored in the FIFO and determines when enough data has been buffered.

The address register increments after every word transfer, and the length register decrements after every word transfer. The address remains constant throughout the burst. Because a transfer is not guaranteed to complete on burst boundaries, additional logic is necessary to recognize the completion of short bursts and complete the transfer.

Related Information

- Avalon Memory-Mapped Master Templates

Reducing Logic Utilization

You can minimize logic size of Qsys systems. Typically, there is a trade-off between logic utilization and performance. Reducing logic utilization applies to both Avalon and AXI interfaces.

Minimizing Interconnect Logic to Reduce Logic Unitization

In Qsys, changes to the connections between master and slave reduce the amount of interconnect logic required in the system.

Related Information

Limited Concurrency on page 8-19

Creating Dedicated Master and Slave Connections to Minimize Interconnect Logic

You can create a system where a master interface connects to a single slave interface. This configuration eliminates address decoding, arbitration, and return data multiplexing, which simplifies the interconnect. Dedicated master-to-slave connections attain the same clock frequencies as Avalon-ST connections.

Typically, these one-to-one connections include an Avalon memory-mapped bridge or hardware accelerator. For example, if you insert a pipeline bridge between a slave and all other master interfaces, the logic between the bridge master and slave interface is reduced to wires. If a hardware accelerator connects only to a dedicated memory, no system interconnect logic is generated between the master and slave pair.

Removing Unnecessary Connections to Minimize Interconnect Logic

The number of connections between master and slave interfaces affects the f_{MAX} of your system. Every master interface that you connect to a slave interface increases the width of the multiplexer width. As a multiplexer width increases, so does the logic depth and width that implements the multiplexer in the FPGA. To improve system performance, connect masters and slaves only when necessary.

When you connect a master interface to many slave interfaces, the multiplexer for the read data signal grows. Avalon typically uses a readdata signal. AXI read data signals add a response status and last indicator to the read response channel using rdata, rresp, and rlast. Additionally, bridges help control the depth of multiplexers.

Related Information

Implementing Command Pipelining (Master-to-Slave) on page 8-12

Simplifying Address Decode Logic

If address code logic is in the critical path, you may be able to change the address map to simplify the decode logic. Experiment with different address maps, including a one-hot encoding, to see if results improve.

Minimizing Arbitration Logic by Consolidating Multiple Interfaces

As the number of components in a design increases, the amount of logic required to implement the interconnect also increases. The number of arbitration blocks increases for every slave interface that is shared by multiple master interfaces. The width of the read data multiplexer increases as the number of slave interfaces supporting read transfers increases on a per master interface basis. For these reasons, consider implementing multiple blocks of logic as a single interface to reduce interconnect logic utilization.

Logic Consolidation Trade-Offs

You should consider the following trade-offs before making modifications to your system or interfaces:

- Consider the impact on concurrency that results when you consolidate components. When a system has four master components and four slave interfaces, it can initiate four concurrent accesses. If you consolidate the four slave interfaces into a single interface, then the four masters must compete for access. Consequently, you should only combine low priority interfaces such as low speed parallel I/O devices if the combination does not impact the performance.
- Determine whether consolidation introduces new decode and multiplexing logic for the slave interface that the interconnect previously included. If an interface contains multiple read and write address locations, the interface already contains the necessary decode and multiplexing logic. When you consolidate interfaces, you typically reuse the decoder and multiplexer blocks already present in one of the original interfaces; however, combining interfaces may simply move the decode and multiplexer logic, rather than eliminate duplication.
- Consider whether consolidating interfaces makes the design complicated. If so, you should not consolidate interfaces.

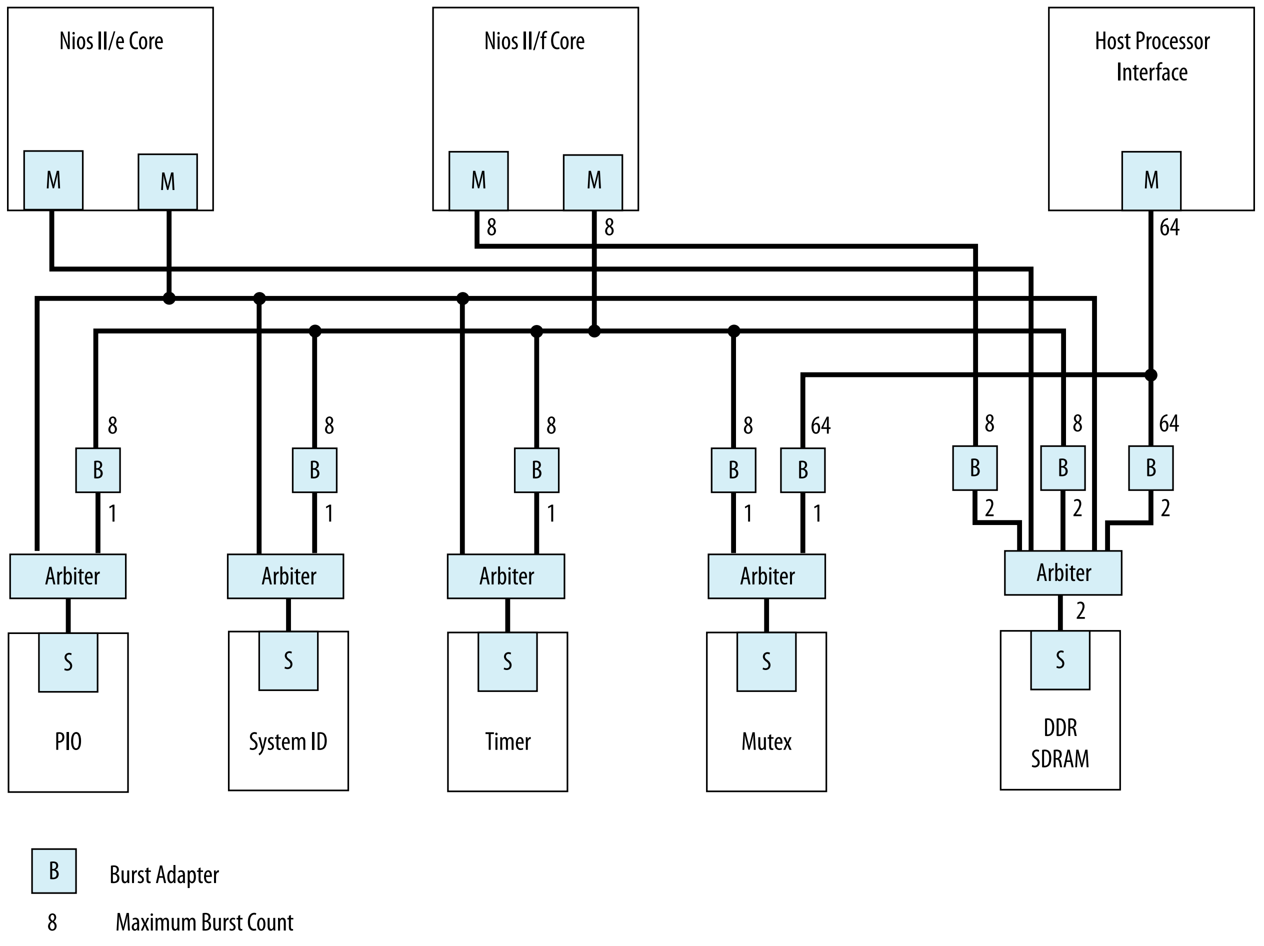
Related Information

Using Concurrency in Memory-Mapped Systems on page 8-6

Consolidating Interfaces

The Nios II/e core maintains communication between the Nios II /f core and external processors. The Nios II/f core supports a maximum burst size of eight. The external processor interface supports a maximum burst length of 64. The Nios II/e core does not support bursting. The memory in the system is SDRAM with an Avalon maximum burst length of two.

Figure 8-21: Mixed Bursting System



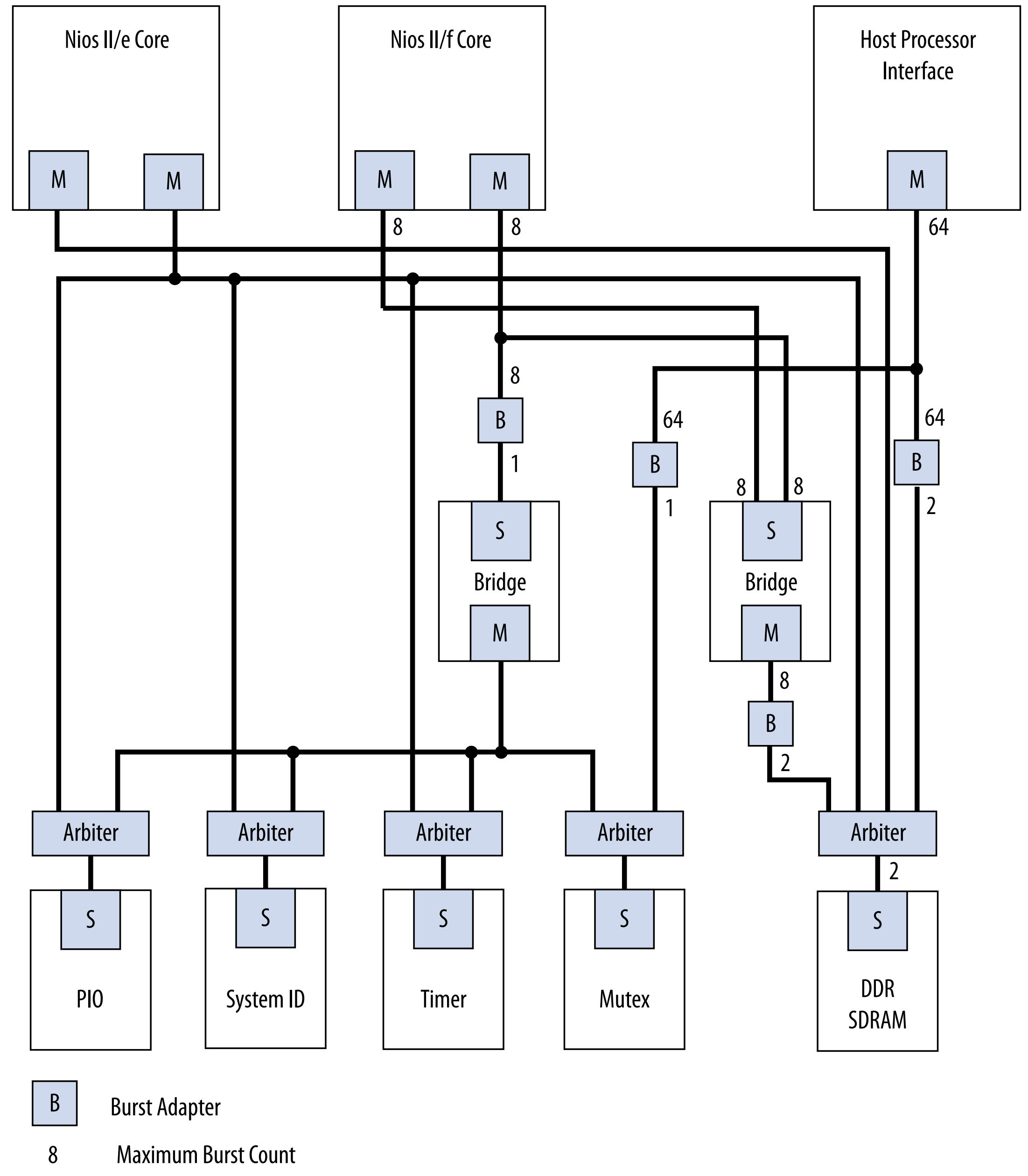
In this example a system with a mix of components with different burst capabilities with a Nios II/e core, a Nios II/f core, and an external processor, which off-loads some processing tasks to the Nios II/f core.

Qsys automatically inserts burst adapters to compensate for burst length mismatches. The adapters reduce bursts to a single transfer, or the length of two transfers. For the external processor interface connecting to DDR SDRAM, a burst of 64 words is divided into 32 burst transfers, each with a burst length of two. When you generate a system, Qsys inserts burst adapters based on maximum `burstcount` values; consequently, the interconnect logic includes burst adapters between masters and slave pairs that do not require bursting, if the master is capable of bursts.

In this example, Qsys inserts a burst adapter between the Nios II processors and the timer, system ID, and PIO peripherals. These components do not support bursting and the Nios II processor performs a single word read and write accesses to these components.

Figure 8-22: Mixed Bursting System with Bridges

To reduce the number of adapters, you can add pipeline bridges. The pipeline bridge, between the Nios II/f core and the peripherals that do not support bursts, eliminates three burst adapters from the previous example. A second pipeline bridge between the Nios II/f core and the DDR SDRAM, with its maximum burst size set to eight, eliminates another burst adapter, as shown below.



Reducing Logic Utilization With Multiple Clock Domains

You specify clock domains in Qsys on the **System Contents** tab. Clock sources can be driven by external input signals to Qsys, or by PLLs inside Qsys. Clock domains are differentiated based on the name of the clock. You can create multiple asynchronous clocks with the same frequency.

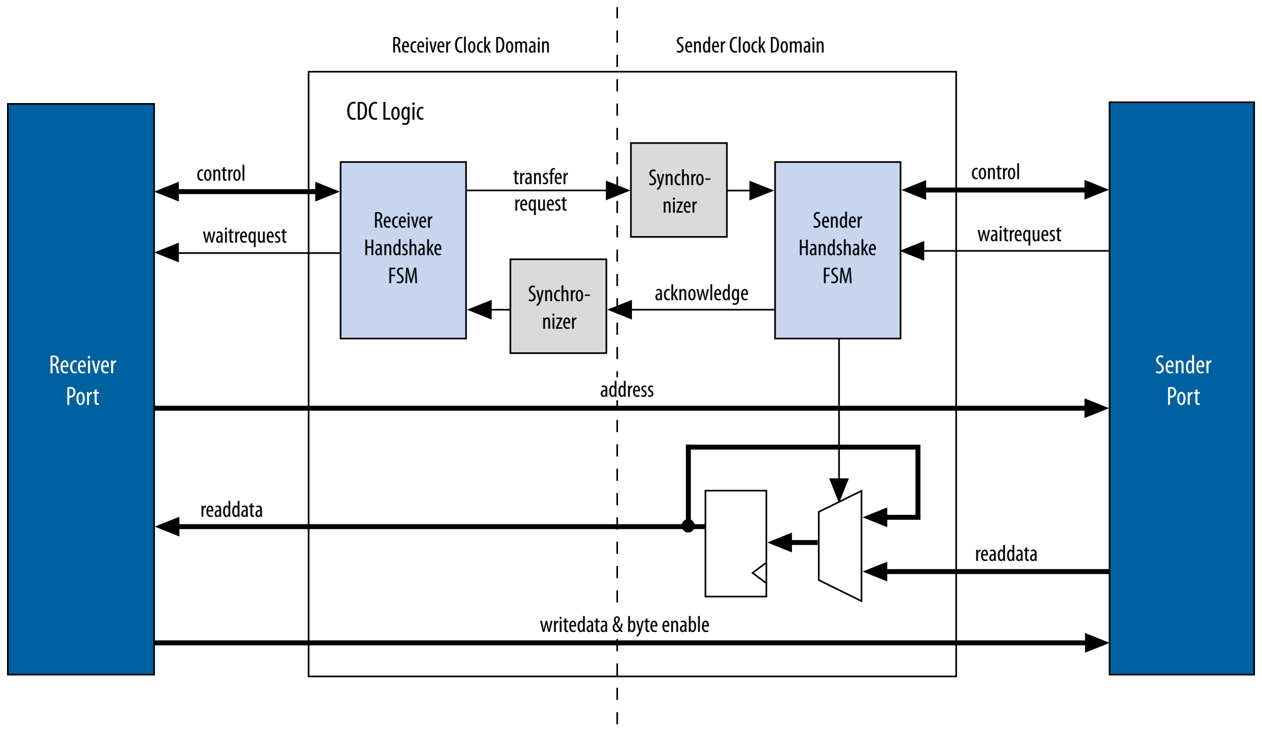
Qsys generates Clock Domain Crossing Logic (CDC) that hides the details of interfacing components operating in different clock domains. The interconnect supports the memory-mapped protocol with each port independently, and therefore masters do not need to incorporate clock adapters in order to interface to slaves on a different domain. Qsys interconnect logic propagates transfers across clock domain boundaries automatically.

Clock-domain adapters provide the following benefits:

- Allows component interfaces to operate at different clock frequencies.
- Eliminates the need to design CDC hardware.
- Allows each memory-mapped port to operate in only one clock domain, which reduces design complexity of components.
- Enables masters to access any slave without communication with the slave clock domain.
- Allows you to focus performance optimization efforts on components that require fast clock speed.

A clock domain adapter consists of two finite state machines (FSM), one in each clock domain, that use a hand-shaking protocol to propagate transfer control signals (`read_request`, `write_request`, and the master `waitrequest` signals) across the clock boundary.

Figure 8-23: Clock Crossing Adapter



This example illustrates a clock domain adapter between one master and one slave. The synchronizer blocks use multiple stages of flip flops to eliminate the propagation of meta-stable events on the control signals that enter the handshake FSMs. The CDC logic works with any clock ratio.

The typical sequence of events for a transfer across the CDC logic is as follows:

- The master asserts address, data, and control signals.
- The master handshake FSM captures the control signals and immediately forces the master to wait. The FSM uses only the control signals, not address and data. For example, the master simply holds the address signal constant until the slave side has safely captured it.
- The master handshake FSM initiates a transfer request to the slave handshake FSM.
- The transfer request is synchronized to the slave clock domain.
- The slave handshake FSM processes the request, performing the requested transfer with the slave.
- When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM. The acknowledge is synchronized back to the master clock domain.
- The master handshake FSM completes the transaction by releasing the master from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave, there is nothing different about a transfer initiated by a master in a different clock domain. From the perspective of a master, a transfer across clock domains simply requires extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay or wait states on the slave side), the Qsys forces the master to wait until the transfer terminates. As a result, pipeline master ports do not benefit from pipelining when performing transfers to a different clock domain.

Qsys automatically determines where to insert CDC logic based on the system and the connections between components, and places CDC logic to maintain the highest transfer rate for all components. Qsys evaluates the need for CDC logic for each master and slave pair independently, and generates CDC logic wherever necessary.

Related Information

Avalon Memory-Mapped Design Optimizations

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case, which is for reads, each transfer is extended by five master clock cycles and five slave clock cycles. Assuming the default value of 2 for the master domain synchronizer length and the slave domain synchronizer length, the components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer.
- Four additional slave clock cycles, due to the slave-side clock synchronizer.
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains.

Note: Systems that require a higher performance clock should use the Avalon-MM clock crossing bridge instead of the automatically inserted CDC logic. The clock crossing bridge includes a buffering mechanism so that multiple reads and writes can be pipelined. After paying the initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of added logic resources.

Reducing Power Consumption

Qsys provides various low power design changes that enable you to reduce the power consumption of the interconnect and custom components.

Reducing Power Consumption With Multiple Clock Domains

When you use multiple clock domains, you should put non-critical logic in the slower clock domain. Qsys automatically reconciles data crossing over asynchronous clock domains by inserting clock crossing logic (handshake or FIFO).

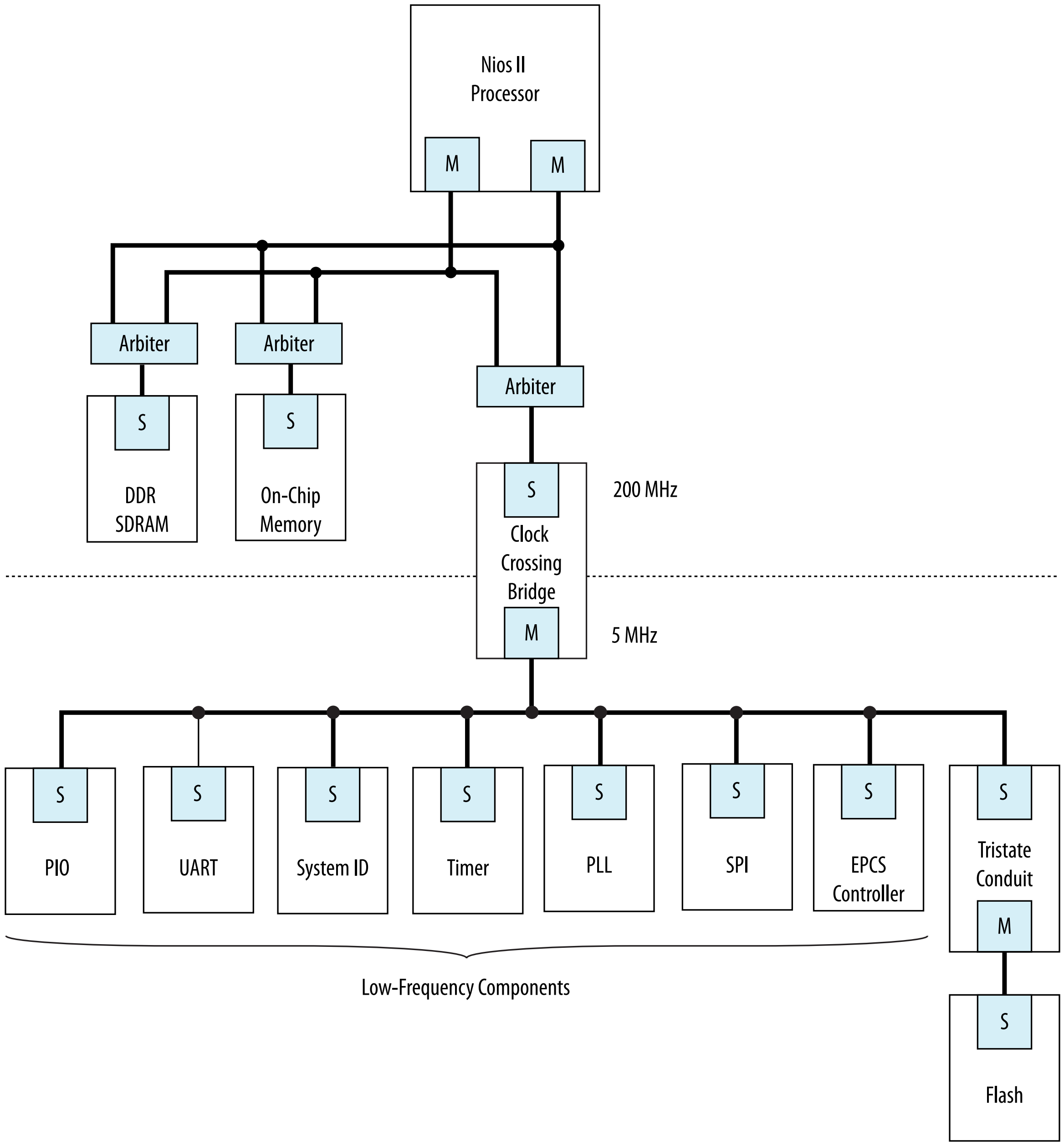
You can use clock crossing in Qsys to reduce the clock frequency of the logic that does not require a high frequency clock, which allows you to reduce power consumption. You can use either handshaking clock crossing bridges or handshaking clock crossing adapters to separate clock domains.

You can use the clock crossing bridge to connect master interfaces operating at a higher frequency to slave interfaces running at a lower frequency. Only connect low throughput or low priority components to a clock crossing bridge that operates at a reduced clock frequency. The following are examples of low throughput or low priority components:

- PIOs
- UARTs (JTAG or RS-232)
- System identification (SysID)
- Timers
- PLL (instantiated within Qsys)
- Serial peripheral interface (SPI)
- EPCS controller
- Tristate bridge and the components connected to the bridge

By reducing the clock frequency of the components connected to the bridge, you reduce the dynamic power consumption of the design. Dynamic power is a function of toggle rates and decreasing the clock frequency decreases the toggle rate.

Figure 8-24: Reducing Power Utilization Using a Bridge to Separate Clock Domains



Qsys automatically inserts clock crossing adapters between master and slave interfaces that operate at different clock frequencies. You can choose the type of clock crossing adapter in the Qsys **Project Settings** tab. Adapters do not appear in the **Connections** column because you do not insert them. The following clock crossing adapter types are available in Qsys:

- **Handshake**—Uses a simple handshaking protocol to propagate transfer control signals and responses across the clock boundary. This adapter uses fewer hardware resources because each transfer is safely propagated to the target domain before the next transfer begins. The Handshake adapter is appropriate for systems with low throughput requirements.
- **FIFO**—Uses dual-clock FIFOs for synchronization. The latency of the FIFO adapter is approximately two clock cycles more than the handshake clock crossing component, but the FIFO-based adapter can sustain higher throughput because it supports multiple transactions simultaneously. The FIFO adapter requires more resources, and is appropriate for memory-mapped transfers requiring high throughput across clock domains.
- **Auto**—Qsys specifies the appropriate FIFO adapter for bursting links and the Handshake adapter for all other links.

Because the clock crossing bridge uses FIFOs to implement the clock crossing logic, it buffers transfers and data. Clock crossing adapters are not pipelined, so that each transaction is blocking until the transaction completes. Blocking transactions may lower the throughput substantially; consequently, if you want to reduce power consumption without limiting the throughput significantly, you should use the clock crossing bridge or the FIFO clock crossing adapter. However, if the design requires single read transfers, a clock crossing adapter is preferable because the latency is lower.

The clock crossing bridge requires few logic resources other than on-chip memory. The number of on-chip memory blocks used is proportional to the address span, data width, buffering depth, and bursting capabilities of the bridge. The clock crossing adapter does not use on-chip memory and requires a moderate number of logic resources. The address span, data width, and the bursting capabilities of the clock crossing adapter determine the resource utilization of the device.

When you decide to use a clock crossing bridge or clock crossing adapter, you must consider the effects of throughput and memory utilization in the design. If on-chip memory resources are limited, you may be forced to choose the clock crossing adapter. Using the clock crossing bridge to reduce the power of a single component may not justify using more resources. However, if you can place all of the low priority components behind a single clock crossing bridge, you may reduce power consumption in the design.

Related Information

Power Optimization

Reducing Power Consumption by Minimizing Toggle Rates

A Qsys system consumes power whenever logic transitions between on and off states. When the state is held constant between clock edges, no charging or discharging occurs. You can use the following design methodologies to reduce the toggle rates of your design:

- Registering component boundaries
- Using clock enable signals
- Inserting bridges

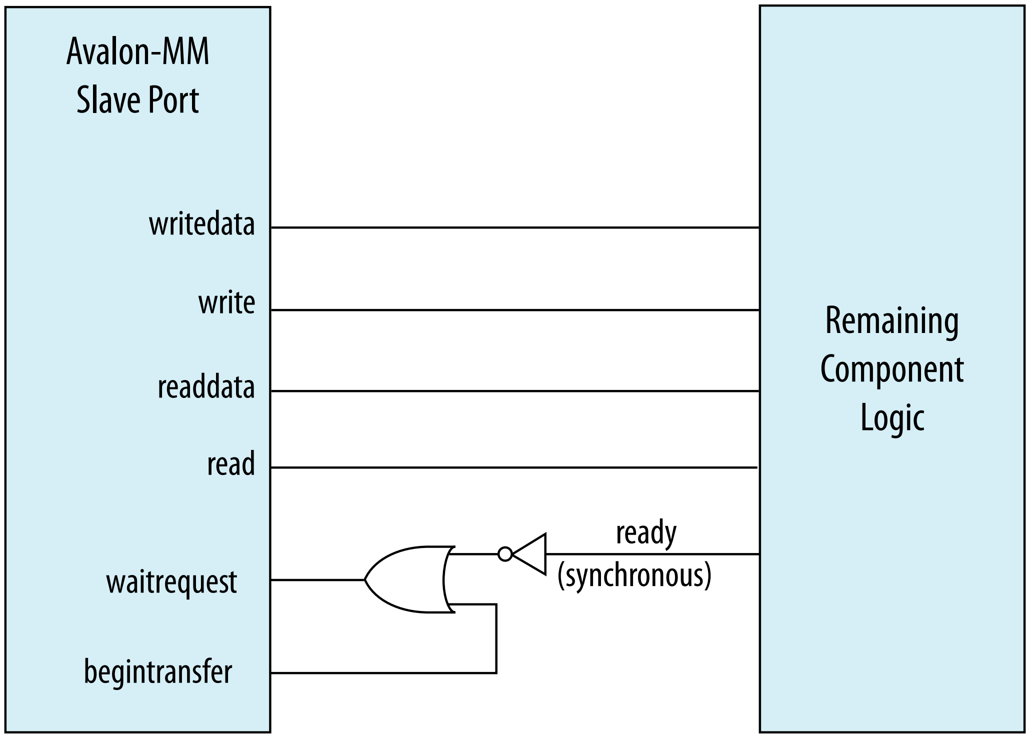
Qsys interconnect is uniquely combinational when no adapters or bridges are present and there is no interconnect pipelining. When a slave interface is not selected by a master, various signals may toggle and propagate into the component. By registering the boundary of your component at the master or slave interface, you can minimize the toggling of the interconnect and your component. In addition, registering boundaries can improve operating frequency. When you register the signals at the interface level, you must ensure that the component continues to operate within the interface standard specification.

Avalon-MM `waitrequest` is a difficult signal to synchronize when you add registers to your component. The `waitrequest` signal must be asserted during the same clock cycle that a master asserts read or write to in order to prolong the transfer. A master interface can read the `waitrequest` signal too early and post more reads and writes prematurely.

Note: There is no direct AXI equivalent for `waitrequest` and `burstcount`, though the *AMBA Protocol Specification* implies that the AXI `ready` signal cannot depend combinatorially on the AXI `valid` signal. Therefore, Qsys typically buffers AXI component boundaries for the `ready` signal.

For slave interfaces, the interconnect manages the `begintransfer` signal, which is asserted during the first clock cycle of any read or write transfer. If the `waitrequest` is one clock cycle late, you can logically `OR` the `waitrequest` and the `begintransfer` signals to form a new `waitrequest` signal that is properly synchronized. Alternatively, the component can assert `waitrequest` before it is selected, guaranteeing that the `waitrequest` is already asserted during the first clock cycle of a transfer.

Figure 8-25: Variable Latency



Using Clock Enables

You can use clock enables to hold the logic in a steady state, and the `write` and `read` signals as clock enables for slave components. Even if you add registers to your component boundaries, the interface can potentially toggle without the use of clock enables. You can also use the clock enable to disable combinational portions of the component.

For example, you can use an active high clock enable to mask the inputs into the combinational logic to prevent it from toggling when the component is inactive. Before preventing inactive logic from toggling, you must determine if the masking causes the circuit to function differently. If masking causes a functional failure, it may be possible to use a register stage to hold the combinational logic constant between clock cycles.

Inserting Bridges

You can use bridges to reduce toggle rates, if you do not want to modify the component by using boundary registers or clock enables. A bridge acts as a repeater where transfers to the slave interface are repeated on the master interface. If the bridge is not accessed, the components connected to its master interface are also not accessed. The master interface of the bridge remains idle until a master accesses the bridge slave interface.

Bridges can also reduce the toggle rates of signals that are inputs to other master interfaces. These signals are typically `readdata`, `readdatavalid`, and `waitrequest`. Slave interfaces that support read accesses drive the `readdata`, `readdatavalid`, and `waitrequest` signals. A bridge inserts either a register or clock crossing FIFO between the slave interface and the master to reduce the toggle rate of the master input signals.

Related Information

- AMBA Protocol Specification
- Power Optimization

Reducing Power Consumption by Disabling Logic

There are typically two types of low power modes: volatile and non-volatile. A volatile low power mode holds the component in a reset state. When the logic is reactivated, the previous operational state is lost. A non-volatile low power mode restores the previous operational state. You can use either software-controlled or hardware-controlled sleep modes to disable a component in order to reduce power consumption.

Software-Controlled Sleep Mode

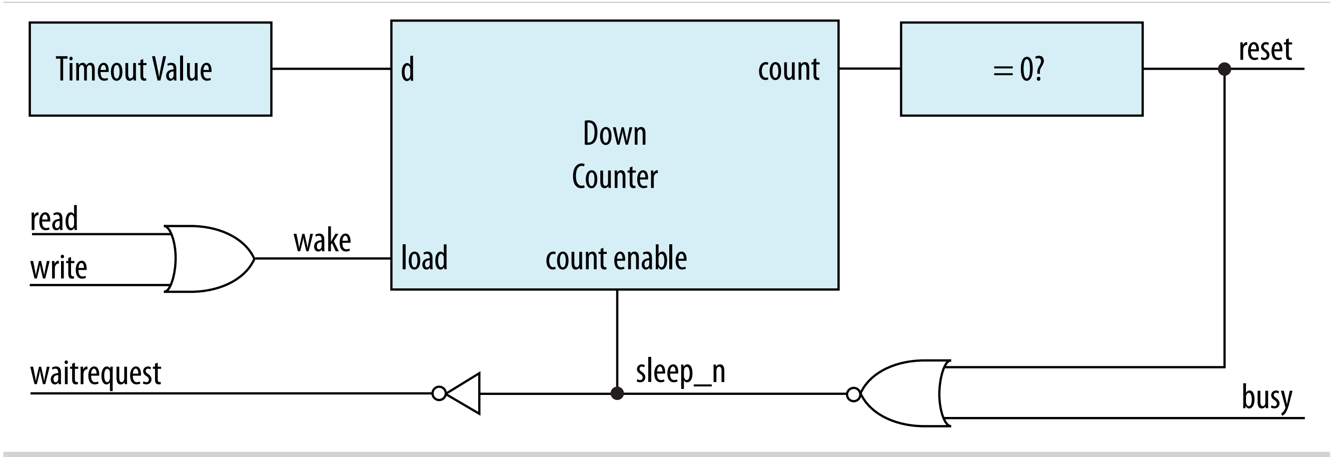
To design a component that supports software-controlled sleep mode, create a single memory-mapped location that enables and disables logic by writing a zero or one. You can use the register's output as a clock enable or reset, depending on whether the component has non-volatile requirements. The slave interface must remain active during sleep mode so that the enable bit is set when the component needs to be activated.

If multiple masters can access a component that supports sleep mode, you can use the mutex core to provide mutually exclusive accesses to your component. You can also build in the logic to re-enable the component on the very first access by any master in your system. If the component requires multiple clock cycles to re-activate, then it must assert a wait request to prolong the transfer as it exits sleep mode.

Hardware-Controlled Sleep Mode

Alternatively, you can implement a timer in your component that automatically causes the component to enter a sleep mode based on a timeout value specified in clock cycles between read or write accesses. Each access resets the timer to the timeout value. Each cycle with no accesses decrements the timeout value by one. If the counter reaches zero, the hardware enters sleep mode until the next access.

Figure 8-26: Hardware-Controlled Sleep Components



This example provides a schematic for the hardware-controlled sleep mode. If restoring the component to an active state takes a long time, use a long timeout value so that the component is not continuously entering and exiting sleep mode. The slave interface must remain functional while the rest of the component is in sleep mode. When the component exits sleep mode, the component must assert the `waitrequest` signal until it is ready for read or write accesses.

Related Information

- Mutex Core
- Power Optimization

Optimizing Qsys System Performance Design Examples

Avalon Pipelined Read Master Example on page 8-41

Multiplexer Examples on page 8-43

Related Information

Avalon Interface Specifications

Avalon Pipelined Read Master Example

For a high throughput system using the Avalon-MM standard, you can design a pipelined read master that allows a system to issue multiple read requests before data returns. Pipelined read masters hide the latency of read operations by posting reads as frequently as every clock cycle. You can use this type of master when the address logic is not dependent on the data returning.

Avalon Pipelined Read Master Example Design Requirements

You must carefully design the logic for the control and data paths of pipelined read masters. The control logic must extend a read cycle whenever the `waitrequest` signal is asserted. This logic must also control the master `address`, `byteenable`, and `read` signals. To achieve maximum throughput, pipelined read masters should post reads continuously as long as `waitrequest` is de-asserted. While `read` is asserted, the address presented to the interconnect is stored.

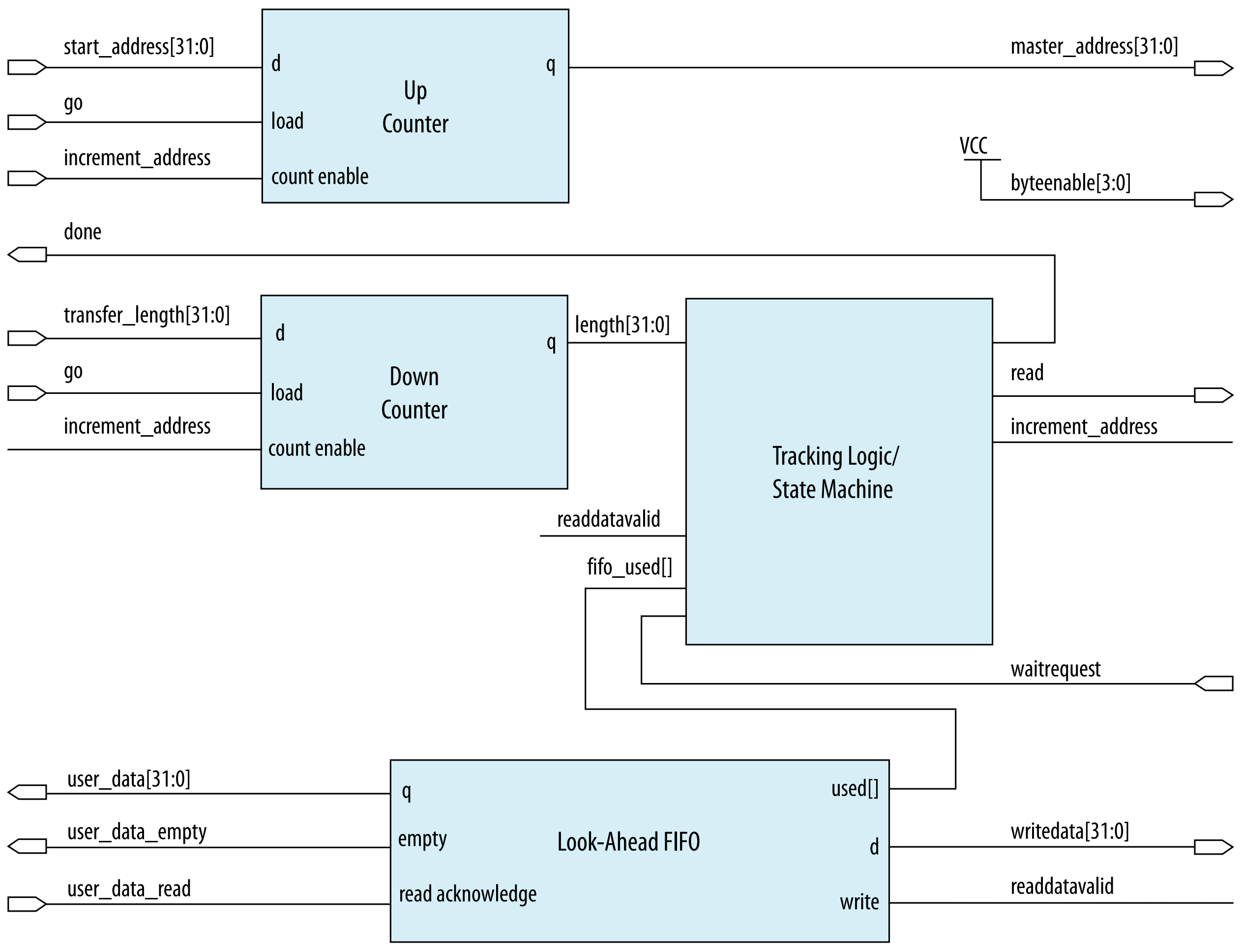
The data path logic includes the `readdata` and `readdatavalid` signals. If your master can accept data on every clock cycle, you can register the data with the `readdatavalid` as an enable bit. If your master cannot process a continuous stream of read data, it must buffer the data in a FIFO. The control logic must stop issuing reads when the FIFO reaches a predetermined fill level to prevent FIFO overflow.

Expected Throughput Improvement

The throughput improvement that you can achieve with a pipelined read master is typically directly proportional to the pipeline depth of the interconnect and the slave interface. For example, if the total latency is two cycles, you can double the throughput by inserting a pipelined read master, assuming the slave interface also supports pipeline transfers. If either the master or slave does not support pipelined read transfers, then the interconnect asserts `waitrequest` until the transfer completes. You can also gain throughput when there are some cycles of overhead before a read response.

Where reads are not pipelined, the throughput is reduced. When both the master and slave interfaces support pipelined read transfers, data flows in a continuous stream after the initial latency. You can use a pipelined read master that stores data in a FIFO to implement a custom DMA, hardware accelerator, or off-chip communication interface.

Figure 8-27: Pipelined Read Master



This example shows a pipelined read master that stores data in a FIFO. The master performs word accesses that are word-aligned and reads from sequential memory addresses. The transfer length is a multiple of the word size.

When the go bit is asserted, the master registers the `start_address` and `transfer_length` signals. The master begins issuing reads continuously on the next clock cycle until the length register reaches zero. In this example, the word size is four bytes so that the address always increments by four, and the length decrements by four. The `read` signal remains asserted unless the FIFO fills to a predetermined level. The address register increments and the length register decrements if the length has not reached 0 and a read is posted.

The master posts a read transfer every time the `read` signal is asserted and the `waitrequest` is deasserted. The master issues reads until the entire buffer has been read or `waitrequest` is asserted. An optional tracking block monitors the done bit. When the length register reaches zero, some reads are outstanding. The tracking logic prevents assertion of done until the last read completes, and monitors the number of reads posted to the interconnect so that it does not exceed the space remaining in the `readdata` FIFO. This example includes a counter that verifies that the following conditions are met:

- If a read is posted and `readdatavalid` is deasserted, the counter increments.
- If a read is not posted and `readdatavalid` is asserted, the counter decrements.

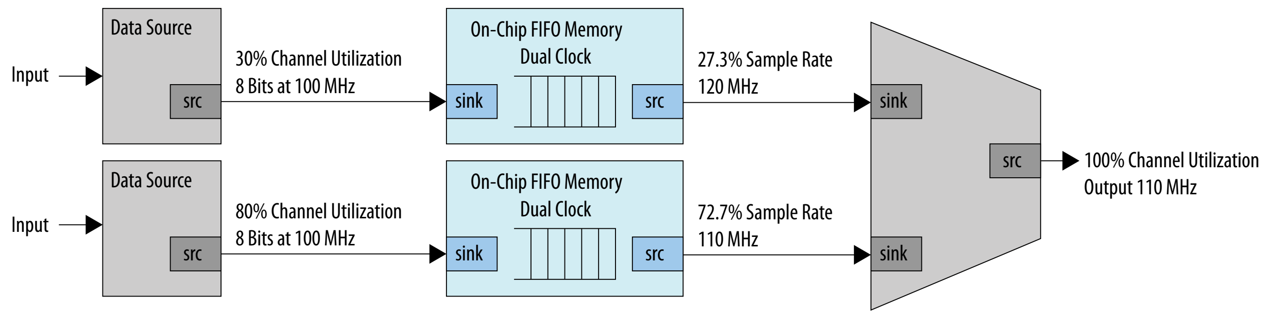
When the `length` register and the tracking logic counter reach zero, all the reads have completed and the done bit is asserted. The `done` bit is important if a second master overwrites the memory locations that the pipelined read master accesses. This bit guarantees that the reads have completed before the original data is overwritten.

Multiplexer Examples

You can combine adapters with streaming components to create data paths whose input and output streams have different properties. The following examples demonstrate datapaths in which the output stream exhibits higher performance than the input stream.

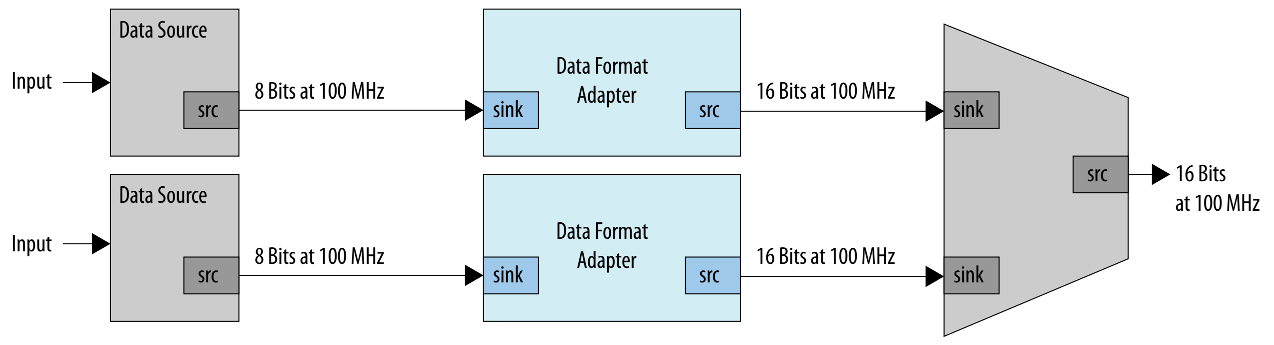
The diagram below illustrates a data path that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. The on-chip FIFO memory has an input clock frequency of 100 MHz, and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time, and the second 72.7 percent of the time. You do not need to know what the typical and maximum input channel utilizations are before for this type of design. For example, if the first channel hits 50% utilization, the output stream exceeds 100% utilization.

Figure 8-28: Data Path that Doubles the Clock Frequency



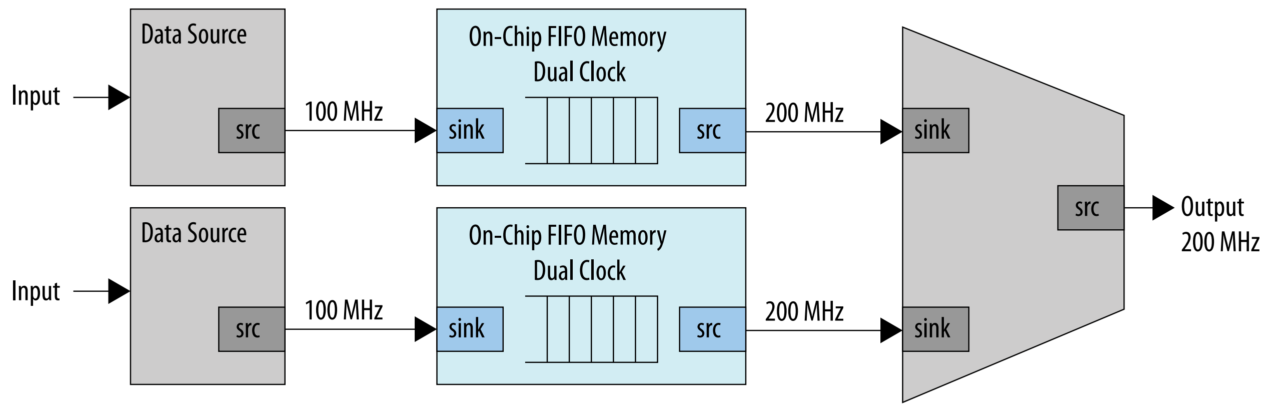
The diagram below illustrates a data path that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. This example shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency.

Figure 8-29: Data Path to Double Data Width and Maintain Original Frequency



The diagram below illustrates a data path that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. This example shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency.

Figure 8-30: Data Path to Boost the Clock Frequency



Document Revision History

The table below indicates edits made to the *Optimizing Qsys System Performance* content since its creation.

Table 8-2: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	*Multiplexer Examples*, rearranged description text for the figures.
May 2013	13.0.0	AMBA APB support.
November 2012	12.1.0	AMBA AXI4 support.
June 2012	12.0.0	AMBA AXI3 support.
November 2011	11.1.0	New document release.

Related Information

Quartus II Handbook Archive

Component Interface Tcl Reference 9

2015.05.04

QII5V1 Subscribe Send Feedback

Tcl commands allow you to perform a wide range of functions in Qsys. Command descriptions contain the Qsys phases where you can use the command, for example, main program, elaboration, composition, or fileset callback.

Qsys supports Avalon, AMBA AXI3 (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB3 (version 1.0) interface specifications.

For more information about procedures for creating IP component **_hw.tcl** files in the Qsys Component Editor, and supported interface standards, refer to *Creating Qsys Components* and *Qsys Interconnect* in volume 1 of the *Quartus II Handbook*.

If you are developing an IP component to work with the Nios II processor, refer to *Publishing Component Information to Embedded Software* in section 3 of the *Nios II Software Developer's Handbook*, which describes how to publish hardware IP component information for embedded software tools, such as a C compiler and a Board Support Package (BSP) generator.

Related Information

- **Avalon Interface Specifications**
- **AMBA Protocol Specifications**
- **Creating Qsys Components** on page 6-1
- **Qsys Interconnect** on page 7-1
- **Publishing Component Information to Embedded Software**

Qsys _hw.tcl Command Reference

To use the current version of the Tcl commands, include the following command at the top of your script:

```
package require -exact qsys <version>
```

Interfaces and Ports

add_interface

Description

Adds an interface to your module. An interface represents a collection of related signals that are managed together in the parent system. These signals are implemented in the IP component's HDL, or exported from an interface from a child instance. As the IP component author, you choose the name of the interface.

Availability

Discovery, Main Program, Elaboration, Composition

Usage

`add_interface` *<name>* *<type>* *<direction>* [*<associated_clock>*]

Returns

No returns value.

Arguments

name

A name you choose to identify an interface.

type

The type of interface.

direction

The interface direction.

associated_clock (optional)

(deprecated) For interfaces requiring associated clocks, use: `set_interface_property` *`<interface>`* `associatedClock` *`<clockInterface>`* For interfaces requiring associated resets, use: `set_interface_property` *`<interface>`* `associatedReset` *`<resetInterface>`*

Example

```
add_interface mm_slave avalon slave

add_interface my_export conduit end
set_interface_property my_export EXPORT_OF uart_0.external_connection
```

Notes

By default, interfaces are enabled. You can set the interface property `ENABLED` to `false` to disable an interface. If an interface is disabled, it is hidden and its ports are automatically terminated to their default values. Active high signals are terminated to 0. Active low signals are terminated to 1.

If the IP component is composed of child instances, the top-level interface is associated with a child instance's interface with `set_interface_property` *`interface`* `EXPORT_OF` *`child_instance.interface`*.

The following direction rules apply to Qsys-supported interfaces.

Interface Type	Direction
avalon	master, slave
axi	master, slave
tristate_conduit	master, slave
avalon_streaming	source, sink
interrupt	sender, receiver
conduit	end
clock	source, sink
reset	source, sink
nios_custom_instruction	slave

Related Information

- **add_interface_port** on page 9-5
- **get_interface_assignments** on page 9-9
- **get_interface_properties** on page 9-11
- **get_interfaces** on page 9-7

add_interface_port

Description

Adds a port to an interface on your module. The name must match the name of a signal on the top-level module in the HDL of your IP component. The port width and direction must be set before the end of the elaboration phase. You can set the port width as follows:

- In the Main program, you can set the port width to a fixed value or a width expression.
- If the port width is set to a fixed value in the Main program, you can update the width in the elaboration callback.

Availability

Main Program, Elaboration

Usage

`add_interface_port` *<interface> <port>* [*<signal_type> <direction> <width_expression>*]

Returns

Arguments

interface

The name of the interface to which this port belongs.

port

The name of the port. This name must match a signal in your top-level HDL for this IP component.

signal_type (optional)

The type of signal for this port, which must be unique. Refer to the *Avalon Interface Specifications* for the signal types available for each interface type.

direction (optional)

The direction of the signal. Refer to *Direction Properties*.

width_expression (optional)

The width of the port, in bits. The width may be a fixed value, or a simple arithmetic expression of parameter values.

Example

```
fixed width:
add_interface_port mm_slave s0_rdata readdata output 32

width expression:
add_parameter DATA_WIDTH INTEGER 32
add_interface_port s0 rdata readdata output "DATA_WIDTH/2"
```

Related Information

- add_interface on page 9-3
- get_port_properties on page 9-13
- get_port_property on page 9-14

- get_port_property on page 9-14
- Direction Properties on page 9-99
- Avalon Interface Specifications

get_interfaces

Description

Returns a list of top-level interfaces.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_interfaces
```

Returns

A list of the top-level interfaces exported from the system.

Arguments

No arguments.

Example

```
get_interfaces
```

Related Information

add_interface on page 9-3

get_interface_assignment

Description

Returns the value of the specified assignment for the specified interface

Availability

Main Program, Elaboration, Validation, Composition

Usage

`get_interface_assignment` *<interface> <assignment>*

Returns

The value of the assignment.

Arguments

interface

The name of a top-level interface.

assignment

The name of an assignment.

Example

```
get_interface_assignment s1 embeddedsw.configuration.isFlash
```

Related Information

- **add_interface** on page 9-3
- **get_interface_assignments** on page 9-9
- **get_interfaces** on page 9-7

get_interface_assignments

Description

Returns the value of all interface assignments for the specified interface.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`get_interface_assignments` *<interface>*

Returns

A list of assignment keys.

Arguments

interface

The name of the top-level interface whose assignment is being retrieved.

Example

```
get_interface_assignments s1
```

Related Information

- **add_interface** on page 9-3
- **get_interface_assignment** on page 9-8
- **get_interfaces** on page 9-7

get_interface_ports

Description

Returns the names of all of the ports that have been added to a given interface. If the interface name is omitted, all ports for all interfaces are returned.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_interface_ports` [<*interface*>]

Returns

A list of port names.

Arguments

interface (optional)

The name of a top-level interface.

Example

```
get_interface_ports mm_slave
```

Related Information

- **add_interface_port** on page 9-5
- **get_port_property** on page 9-14
- **set_port_property** on page 9-18

get_interface_properties

Description

Returns the names of all the interface properties for the specified interface as a space separated list

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_interface_properties` *<interface>*

Returns

A list of properties for the interface.

Arguments

interface

The name of an interface.

Example

```
get_interface_properties interface
```

Notes

The properties for each interface type are different. Refer to the *Avalon Interface Specifications* for more information about interface properties.

Related Information

- **get_interface_property** on page 9-12
- **set_interface_property** on page 9-17
- **Avalon Interface Specifications**

get_interface_property

Description

Returns the value of a single interface property from the specified interface.

Availability

Discovery, Main Program, Elaboration, Composition, Fileset Generation

Usage

`get_interface_property` *<interface> <property>*

Returns

Arguments

interface

The name of an interface.

property

The name of the property whose value you want to retrieve. Refer to *Interface Properties*.

Example

```
get_interface_property mm_slave linewrapBursts
```

Notes

The properties for each interface type are different. Refer to the *Avalon Interface Specifications* for more information about interface properties.

Related Information

- get_interface_properties on page 9-11
- set_interface_property on page 9-17
- Avalon Interface Specifications

get_port_properties

Description

Returns a list of port properties.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_port_properties
```

Returns

A list of port properties. Refer to *Port Properties*.

Arguments

No arguments.

Example

```
get_port_properties
```

Related Information

- **add_interface_port** on page 9-5
- **get_port_property** on page 9-14
- **set_port_property** on page 9-18
- **Port Properties** on page 9-97

get_port_property

Description

Returns the value of a property for the specified port.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_port_property` *<port> <property>*

Returns

The value of the property.

Arguments

port

The name of the port.

property

The name of a port property. Refer to *Port Properties*.

Example

```
get_port_property rdata WIDTH_VALUE
```

Related Information

- **add_interface_port** on page 9-5
- **get_port_properties** on page 9-13
- **set_port_property** on page 9-18
- **Port Properties** on page 9-97

set_interface_assignment

Description

Sets the value of the specified assignment for the specified interface.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`set_interface_assignment` *<interface> <assignment>* [*<value>*]

Returns

No return value.

Arguments

interface

The name of the top-level interface whose assignment is being set.

assignment

The assignment whose value is being set.

value (optional)

The new assignment value.

Example

```
set_interface_assignment s1 embeddedsw.configuration.isFlash 1
```

Notes

Assignments for Nios II Software Build Tools

Interface assignments provide extra data for the Nios II Software Build Tools working with the generated system.

Assignments for Qsys Tools

There are several assignments that guide behavior in the Qsys tools.

`qsys.ui.export_name:`	If present, this interface should always be exported when an instance is added to a Qsys system. The value is the requested name of the exported interface in the parent system.
`qsys.ui.connect:`	If present, this interface should be auto-connected when an instance is added to a Qsys system. The value is a comma-separated list of other interfaces on the same instance that should be connected with this interface.
`ui.blockdiagram.direction:`	If present, the direction of this interface in the block diagram is set by the user. The value is either "output" or "input".

Related Information

- **add_interface** on page 9-3
- **get_interface_assignment** on page 9-8
- **get_interface_assignments** on page 9-9

set_interface_property

Description

Sets the value of a property on an exported top-level interface. You can use this command to set the `EXPORT_OF` property to specify which interface of a child instance is exported via this top-level interface.

Availability

Main Program, Elaboration, Composition

Usage

`set_interface_property` *<interface> <property> <value>*

Returns

No return value.

Arguments

interface

The name of an exported top-level interface.

property

The name of the property Refer to *Interface Properties*.

value

The new property value.

Example

```
set_interface_property clk_out EXPORT_OF clk.clk_out
set_interface_property mm_slave linewrapBursts false
```

Notes

The properties for each interface type are different. Refer to the *Avalon Interface Specifications* for more information about interface properties.

Related Information

- **get_interface_properties** on page 9-11
- **get_interface_property** on page 9-12
- **Interface Properties** on page 9-90
- **Avalon Interface Specifications**

set_port_property

Description

Sets a port property.

Availability

Main Program, Elaboration

Usage

`set_port_property` *<port> <property>* [*<value>*]

Returns

The new value.

Arguments

port

The name of the port.

property

One of the supported properties. Refer to *Port Properties*.

value (optional)

The value to set.

Example

```
set_port_property rdata WIDTH 32
```

Related Information

- **add_interface_port** on page 9-5
- **get_port_properties** on page 9-13
- **set_port_property** on page 9-18

set_interface_upgrade_map

Description

Maps the interface name of an older version of an IP core to the interface name of the current IP core. The interface type must be the same between the older and newer versions of the IP cores. This allows system connections and properties to maintain proper functionality. By default, if the older and newer versions of IP core have the same name and type, then Qsys maintains all properties and connections automatically.

Availability

Parameter Upgrade

Usage

```
set_interface_upgrade_map { <old_interface_name> <new_interface_name>
<old_interface_name_2> <new_interface_name_2> … }
```

Returns

No return value.

Arguments

{ <old_interface_name> <new_interface_name>}

List of mappings between between names of older and newer interfaces.

Example

```
set_interface_upgrade_map { avalon_master_interface new_avalon_master_interface }
```

Parameters

add_parameter

Description

Adds a parameter to your IP component.

Availability

Main Program

Usage

`add_parameter` *<name>* *<type>* [*<default_value>* *<description>*]

Returns

Arguments

name

The name of the parameter.

type

The data type of the parameter Refer to *Parameter Type Properties*.

default_value (optional)

The initial value of the parameter in a new instance of the IP component.

description (optional)

Explains the use of the parameter.

Example

```
add_parameter seed INTEGER 17 "The seed to use for data generation."
```

Notes

Most parameter types have a single GUI element for editing the parameter value. `string_list` and `integer_list` parameters are different, because they are edited as tables. A multi-column table can be created by grouping multiple into a single table. To edit multiple list parameters in a single table, the display items for the parameters must be added to a group with a `TABLE` hint:

```
add_parameter coefficients INTEGER_LIST add_parameter positions INTEGER_LIST
add_display_item "" "Table Group" GROUP TABLE add_display_item "Table Group"
coefficients PARAMETER add_display_item "Table Group" positions PARAMETER
```

Related Information

- **get_parameter_properties** on page 9-23
- **get_parameter_property** on page 9-24
- **get_parameter_value** on page 9-25
- **set_parameter_property** on page 9-29
- **set_parameter_value** on page 9-30
- **Parameter Type Properties** on page 9-95

get_parameters

Description

Returns the names of all the parameters in the IP component.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_parameters
```

Returns

A list of parameter names

Arguments

No arguments.

Example

```
get_parameters
```

Related Information

- **add_parameter** on page 9-21
- **get_parameter_property** on page 9-24
- **get_parameter_value** on page 9-25
- **get_parameters** on page 9-22
- **set_parameter_property** on page 9-29

get_parameter_properties

Description

Returns a list of all the parameter properties as a list of strings. The `get_parameter_property` and `set_parameter_property` commands are used to get and set the values of these properties, respectively.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_parameter_properties
```

Returns

A list of parameter property names. Refer to *Parameter Properties*.

Arguments

No arguments.

Example

```
set property_summary [ get_parameter_properties ]
```

Related Information

- **add_parameter** on page 9-21
- **get_parameter_property** on page 9-24
- **get_parameter_value** on page 9-25
- **get_parameters** on page 9-22
- **set_parameter_property** on page 9-29
- **Parameter Properties** on page 9-92

get_parameter_property

Description

Returns the value of a property of a parameter.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_parameter_property` *<parameter> <property>*

Returns

The value of the property.

Arguments

parameter

The name of the parameter whose property value is being retrieved.

property

The name of the property. Refer to *Parameter Properties*.

Example

```
set enabled [ get_parameter_property parameter1 ENABLED ]
```

Related Information

- **add_parameter** on page 9-21
- **get_parameter_properties** on page 9-23
- **get_parameter_value** on page 9-25
- **get_parameters** on page 9-22
- **set_parameter_property** on page 9-29
- **set_parameter_value** on page 9-30
- **Parameter Properties** on page 9-92

get_parameter_value

Description

Returns the current value of a parameter defined previously with the `add_parameter` command.

Availability

Discovery, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_parameter_value` <*parameter*>

Returns

The value of the parameter.

Arguments

parameter

The name of the parameter whose value is being retrieved.

Example

```
set width [ get_parameter_value fifo_width ]
```

Notes

If `AFFECTS_ELABORATION` is `false` for a given parameter, `get_parameter_value` is not available for that parameter from the elaboration callback. If `AFFECTS_GENERATION` is `false` then it is not available from the generation callback.

Related Information

- **add_parameter** on page 9-21
- **get_parameter_property** on page 9-24
- **get_parameters** on page 9-22
- **set_parameter_property** on page 9-29
- **set_parameter_value** on page 9-30

get_string

Description

Returns the value of an externalized string previously loaded by the `load_strings` command.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_string` *<identifier>*

Returns

The externalized string.

Arguments

identifier

The string identifer.

Example

```
hw.tcl:
load_strings test.properties
set_module_property NAME test
set_module_property VERSION [get_string VERSION]
set_module_property DISPLAY_NAME [get_string DISPLAY_NAME]
add_parameter firepower INTEGER 0 ""
set_parameter_property firepower DISPLAY_NAME [get_string PARAM_DISPLAY_NAME]
set_parameter_property firepower TYPE INTEGER
set_parameter_property firepower DESCRIPTION [get_string PARAM_DESCRIPTION]

test.properties:
DISPLAY_NAME = Trogdor!
VERSION = 1.0
PARAM_DISPLAY_NAME = Firepower
PARAM_DESCRIPTION = The amount of force to use when breathing fire.
```

Notes

Use uppercase words separated with underscores to name string identifiers. If you are externalizing module properties, use the module property name for the string identifier:

```
set_module_property DISPLAY_NAME [get_string DISPLAY_NAME]
```

If you are externalizing a parameter property, qualify the parameter property with the parameter name, with uppercase format, if needed:

```
set_parameter_property my_param DISPLAY_NAME [get_string MY_PARAM_DISPLAY_NAME]
```

If you use a string to describe a string format, end the identifier with _FORMAT.

```
set formatted_string [ format  [ get_string TWO_ARGUMENT_MESSAGE_FORMAT ] "arg1"
"arg2" ]
```

Related Information

load_strings on page 9-28

load_strings

Description

Loads strings from an external `.properties` file.

Availability

Discovery, Main Program

Usage

`load_strings` *<path>*

Returns

No return value.

Arguments

path

The path to the properties file.

Example

```
hw.tcl:
load_strings test.properties
set_module_property NAME test
set_module_property VERSION [get_string VERSION]
set_module_property DISPLAY_NAME [get_string DISPLAY_NAME]
add_parameter firepower INTEGER 0 ""
set_parameter_property firepower DISPLAY_NAME [get_string PARAM_DISPLAY_NAME]
set_parameter_property firepower TYPE INTEGER
set_parameter_property firepower DESCRIPTION [get_string PARAM_DESCRIPTION]

test.properties:
DISPLAY_NAME = Trogdor!
VERSION = 1.0
PARAM_DISPLAY_NAME = Firepower
PARAM_DESCRIPTION = The amount of force to use when breathing fire.
```

Notes

Refer to the *Java Properties File* for properties file format. A `.properties` file is a text file with *`KEY`*=*`value`* pairs. For externalized strings, the *`KEY`* is a string identifier and the *`value`* is the externalized string.

For example:

```
TROGDOR = A dragon with a big beefy arm
```

Related Information

- **get_string** on page 9-26
- **Java Properties File**

set_parameter_property

Description

Sets a single parameter property.

Availability

Main Program, Edit, Elaboration, Validation, Composition

Usage

`set_parameter_property` *<parameter> <property> <value>*

Returns

Arguments

parameter

The name of the parameter that is being set.

property

The name of the property. Refer to *Parameter Properties*.

value

The new value for the property.

Example

```
set_parameter_property BAUD_RATE ALLOWED_RANGES {9600 19200 38400}
```

Related Information

- **add_parameter** on page 9-21
- **get_parameter_properties** on page 9-23
- **set_parameter_property** on page 9-29
- **Parameter Properties** on page 9-92

set_parameter_value

Description

Sets a parameter value. The value of a derived parameter can be updated by the IP component in the elaboration callback or the edit callback. Any changes to the value of a derived parameter in the edit callback will not be preserved.

Availability

Edit, Elaboration, Validation, Composition, Parameter Upgrade

Usage

`set_parameter_value` *<parameter> <value>*

Returns

No return value.

Arguments

parameter

The name of the parameter that is being set.

value

Specifies the new parameter value.

Example

```
set_parameter_value half_clock_rate [ expr { [ get_parameter_value clock_rate ] /
2 } ]
```

decode_address_map

Description

Converts an XML–formatted address map into a list of Tcl lists. Each inner list is in the correct format for conversion to an array. The XML code that describes each slave includes: its name, start address, and end address.

Availability

Elaboration, Generation, Composition

Usage

`decode_address_map` *<address_map_XML_string>*

Returns

No return value.

Arguments

address_mapXML_string

An XML string that describes the address map of a master.

Example

In this example, the code describes the address map for the master that accesses the `ext_ssram`, `sys_clk_timer` and `sysid` slaves. The format of the string may differ from the example below; it may have different white space between the elements and include additional attributes or elements. Use the `decode_address_map` command to decode the code that represents a master's address map to ensure that your code works with future versions of the address map.

```
<address-map>
  <slave name='ext_ssram' start='0x01000000' end='0x01200000' />
  <slave name='sys_clk_timer' start='0x02120800' end='0x02120820' />
  <slave name='sysid' start='0x021208B8' end='0x021208C0' />
</address-map>
```

Note: Altera recommends that you use the code provided below to enumerate over the IP components within an address map, rather than writing your own parser.

```
set address_map_xml [get_parameter_value my_map_param]
set address_map_dec [decode_address_map $address_map_xml]
foreach i $address_map_dec {
    array set info $i
    send_message info "Connected to slave $info(name)"
}
```

Display Items

add_display_item

Description

Specifies the following aspects of the IP component display:

- Creates logical groups for a IP component's parameters. For example, to create separate groups for the IP component's timing, size, and simulation parameters. An IP component displays the groups and parameters in the order that you specify the display items in the **_hw.tcl** file.
- Groups a list of parameters to create multi-column tables.
- Specifies an image to provide representation of a parameter or parameter group.
- Creates a button by adding a display item of type `action`. The display item includes the name of the callback to run.

Availability

Main Program

Usage

`add_display_item` <*parent_group*> <*id*> <*type*> [<*args*>]

Returns

Arguments

parent_group

Specifies the group to which a display item belongs

id

The identifier for the display item. If the item being added is a parameter, this is the parameter name. If the item is a group, this is the group name.

type

The type of the display item. Refer to *Display Item Kind Properties*.

args (optional)

Provides extra information required for display items.

Example

```
add_display_item "Timing" read_latency PARAMETER
add_display_item "Sounds" speaker_image_id ICON speaker.jpg
```

Notes

The following examples illustrate further illustrate the use of arguments:

- `add_display_item groupName id icon` *`path-to-image-file`*
- `add_display_item groupName parameterName parameter`
- `add_display_item groupName id text "your-text"`

  The your-text argument is a block of text that is displayed in the GUI. Some simple HTML formatting is allowed, such as `<b>` and `<i>`, if the text starts with `<html>`.
- `add_display_item parentGroupName childGroupName group [tab]`

  The tab is an optional parameter. If present, the group appears in separate tab in the GUI for the instance.
- `add_display_item parentGroupName actionName action buttonClickCallbackProc`

Related Information

- **get_display_item_properties** on page 9-36
- **get_display_item_property** on page 9-37
- **get_display_items** on page 9-35
- **set_display_item_property** on page 9-38
- **Display Item Kind Properties** on page 9-101

get_display_items

Description

Returns a list of all items to be displayed as part of the parameterization GUI.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_display_items
```

Returns

List of display item IDs.

Arguments

No arguments.

Example

```
get_display_items
```

Related Information

- **add_display_item** on page 9-33
- **get_display_item_properties** on page 9-36
- **get_display_item_property** on page 9-37
- **set_display_item_property** on page 9-38

get_display_item_properties

Description

Returns a list of names of the properties of display items that are part of the parameterization GUI.

Availability

Main Program

Usage

```
get_display_item_properties
```

Returns

A list of display item property names. Refer to *Display Item Properties*.

Arguments

No arguments.

Example

```
get_display_item_properties
```

Related Information

- **add_display_item** on page 9-33
- **get_display_item_property** on page 9-37
- **set_display_item_property** on page 9-38
- **Display Item Properties** on page 9-100

get_display_item_property

Description

Returns the value of a specific property of a display item that is part of the parameterization GUI.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`get_display_item_property` *<display_item> <property>*

Returns

The value of a display item property.

Arguments

display_item

The id of the display item.

property

The name of the property. Refer to *Display Item Properties*.

Example

```
set my_label [get_display_item_property my_action DISPLAY_NAME]
```

Related Information

- **add_display_item** on page 9-33
- **get_display_item_properties** on page 9-36
- **get_display_items** on page 9-35
- **set_display_item_property** on page 9-38
- **Display Item Properties** on page 9-100

set_display_item_property

Description

Sets the value of specific property of a display item that is part of the parameterization GUI.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Composition

Usage

`set_display_item_property` *<display_item> <property> <value>*

Returns

No return value.

Arguments

display_item

The name of the display item whose property value is being set.

property

The property that is being set. Refer to *Display Item Properties*.

value

The value to set.

Example

```
set_display_item_property my_action DISPLAY_NAME "Click Me"
set_display_item_property my_action DESCRIPTION "clicking this button runs the
click_me_callback proc in the hw.tcl file"
```

Related Information

- **add_display_item** on page 9-33
- **get_display_item_properties** on page 9-36
- **get_display_item_property** on page 9-37
- **Display Item Properties** on page 9-100

Module Definition

add_documentation_link

Description

Allows you to link to documentation for your IP component.

Availability

Discovery, Main Program

Usage

`add_documentation_link` <*title*> <*path*>

Returns

No return value.

Arguments

title

The title of the document for use on menus and buttons.

path

A path to the IP component documentation, using a syntax that provides the entire URL, not a relative path. For example: `http://www.mydomain.com/my_memory_controller.html` or `file:///datasheet.txt`

Example

```
add_documentation_link "Avalon Verification IP Suite User Guide" http://
www.altera.com/literature/ug/ug_avalon_verification_ip.pdf
```

get_module_assignment

Description

This command returns the value of an assignment. You can use the `get_module_assignment` and `set_module_assignment` and the `get_interface_assignment` and `set_interface_assignment` commands to provide information about the IP component to embedded software tools and applications.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`get_module_assignment` <*assignment*>

Returns

The value of the assignment

Arguments

assignment

The name of the assignment whose value is being retrieved

Example

```
get_module_assignment embeddedsw.CMacro.colorSpace
```

Related Information

- **get_module_assignments** on page 9-42
- **set_module_assignment** on page 9-47

get_module_assignments

Description

Returns the names of the module assignments.

Availability

Main Program, Elaboration, Validation, Composition

Usage

```
get_module_assignments
```

Returns

A list of assignment names.

Arguments

No arguments.

Example

```
get_module_assignments
```

Related Information

- **get_module_assignment** on page 9-41
- **set_module_assignment** on page 9-47

get_module_ports

Description

Returns a list of the names of all the ports which are currently defined.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_module_ports
```

Returns

A list of port names.

Arguments

No arguments.

Example

```
get_module_ports
```

Related Information

- **add_interface** on page 9-3
- **add_interface_port** on page 9-5

get_module_properties

Description

Returns the names of all the module properties as a list of strings. You can use the `get_module_property` and `set_module_property` commands to get and set values of individual properties. The value returned by this command is always the same for a particular version of Qsys

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_module_properties
```

Returns

List of strings. Refer to *Module Properties*.

Arguments

No arguments.

Example

```
get_module_properties
```

Related Information

- **get_module_property** on page 9-45
- **set_module_property** on page 9-48
- **Module Properties** on page 9-103

get_module_property

Description

Returns the value of a single module property.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_module_property` *<property>*

Returns

Various.

Arguments

property

The name of the property, Refer to *Module Properties*.

Example

```
set my_name [ get_module_property NAME ]
```

Related Information

- **get_module_properties** on page 9-44
- **set_module_property** on page 9-48
- **Module Properties** on page 9-103

send_message

Description

Sends a message to the user of the IP component. The message text is normally interpreted as HTML. You can use the `<b>` element to provide emphasis. If you do not want the message text to be interpreted as HTML, then pass a list as the message level, for example, `{ Info Text }`.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`send_message` *<level> <message>*

Returns

No return value .

Arguments

level

The following message levels are supported:

- **`ERROR`**--Provides an error message. The Qsys system cannot be generated with existing error messages.
- **`WARNING`**--Provides a warning message.
- **`INFO`**--Provides an informational message.
- **`PROGRESS`**--Reports progress during generation.
- **`DEBUG`**--Provides a debug message when debug mode is enabled.

message

The text of the message.

Example

```
send_message ERROR "The system is down!"
send_message { Info Text } "The system is up!"
```

set_module_assignment

Description

Sets the value of the specified assignment.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`set_module_assignment` *<assignment>* [<*value*>]

Returns

No return value.

Arguments

assignment

The assignment whose value is being set

value (optional)

The value of the assignment

Example

```
set_module_assignment embeddedsw.CMacro.colorSpace CMYK
```

Related Information

- **get_module_assignment** on page 9-41
- **get_module_assignments** on page 9-42

set_module_property

Description

Allows you to set the values for module properties.

Availability

Discovery, Main Program

Usage

`set_module_property` *<property> <value>*

Returns

No return value.

Arguments

property

The name of the property. Refer to *Module Properties*.

value

The new value of the property.

Example

```
set_module_property VERSION 10.0
```

Related Information

- **get_module_properties** on page 9-44
- **get_module_property** on page 9-45
- **Module Properties** on page 9-103

add_hdl_instance

Description

Adds an instance of a predefined module, referred to as a *child* or *child instance*. The HDL entity generated from this instance can be instantiated and connected within this IP component's HDL.

Availability

Main Program, Elaboration, Composition

Usage

`add_hdl_instance` *<entity_name> <ip_core_type>* [*<version>*]

Returns

The entity name of the added instance.

Arguments

entity_name

Specifies a unique local name that you can use to manipulate the instance. This name is used in the generated HDL to identify the instance.

ip_core_type

The type refers to a kind of instance available in the IP Catalog, for example `altera_avalon_uart`.

version (optional)

The required version of the specified instance type. If no version is specified, the latest version is used.

Example

```
add_hdl_instance my_uart altera_avalon_uart
```

Related Information

- **get_instance_parameter_value** on page 9-67
- **get_instance_parameters** on page 9-65
- **get_instances** on page 9-57
- **set_instance_parameter_value** on page 9-70

package

Description

Allows you to specify a particular version of the Qsys software to avoid software compatibility issues, and to determine which version of the **_hw.tcl** API to use for the IP component. You must use the package command at the beginning of your **_hw.tcl** file.

Availability

Main Program

Usage

`package require -exact qsys` *<version>*

Returns

No return value

Arguments

version

The version of Qsys that you require, such as 14.1.

Example

```
package require -exact qsys 14.1
```

Composition

add_instance

Description

Adds an instance of an IP component, referred to as a child or child instance to the subsystem. You can use this command to create IP components that are composed of other IP component instances. The HDL for this subsystem will be generated; no custom HDL will need to be written for the IP component.

Availability

Main Program, Composition

Usage

`add_instance` *<name> <type>* [*<version>*]

Returns

No return value.

Arguments

name

Specifies a unique local name that you can use to manipulate the instance. This name is used in the generated HDL to identify the instance.

type

The type refers to a type available in the IP Catalog, for example `altera_avalon_uart`.

version (optional)

The required version of the specified type. If no version is specified, the highest available version is used.

Example

```
add_instance my_uart altera_avalon_uart
add_instance my_uart altera_avalon_uart 14.1
```

Related Information

- **add_connection** on page 9-52
- **get_instance_interface_property** on page 9-64
- **get_instance_parameter_value** on page 9-67
- **get_instance_parameters** on page 9-65
- **get_instance_property** on page 9-61
- **get_instances** on page 9-57
- **set_instance_parameter_value** on page 9-70

add_connection

Description

Connects the named interfaces on child instances together using an appropriate connection type. Both interface names consist of a child instance name, followed by the name of an interface provided by that module. For example, `mux0.out` is the interface named `out` on the instance named `mux0`. Be careful to connect the start to the end, and not the other way around.

Availability

Main Program, Composition

Usage

`add_connection` *<start>* [*<end>* *<kind>* *<name>*]

Returns

The name of the newly added connection in `start.point/end.point` format.

Arguments

start

The start interface to be connected, in `<instance_name>.<interface_name>` format.

end (optional)

The end interface to be connected, `<instance_name>.<interface_name>`.

kind (optional)

The type of connection, such as `avalon` or `clock`.

name (optional)

A custom name for the connection. If unspecified, the name will be `<start_instance>.<interface>.<end_instance><interface>`

Example

```
add_connection dma.read_master sdram.s1 avalon
```

Related Information

- add_instance on page 9-52
- get_instance_interfaces on page 9-58

get_connections

Description

Returns a list of all connections in the composed subsystem.

Availability

Main Program, Composition

Usage

```
get_connections
```

Returns

A list of connections.

Arguments

No arguments.

Example

```
set all_connections [ get_connections ]
```

Related Information

add_connection on page 9-52

get_connection_parameters

Description

Returns a list of parameters found on a connection.

Availability

Main Program, Composition

Usage

`get_connection_parameters` *<connection>*

Returns

A list of parameters names

Arguments

connection

The connection to query.

Example

```
get_connection_parameters cpu.data_master/dma0.csr
```

Related Information

- **add_connection** on page 9-52
- **get_connection_parameter_value** on page 9-56

get_connection_parameter_value

Description

Returns the value of a parameter on the connection. Parameters represent aspects of the connection that can be modified once the connection is created, such as the base address for an Avalon Memory Mapped connection.

Availability

Composition

Usage

`get_connection_parameter_value` *<connection> <parameter>*

Returns

The value of the parameter.

Arguments

connection

The connection to query.

parameter

The name of the parameter.

Example

```
get_connection_parameter_value cpu.data_master/dma0.csr baseAddress
```

Related Information

- **add_connection** on page 9-52
- **get_connection_parameters** on page 9-55

get_instances

Description

Returns a list of the instance names for all child instances in the system.

Availability

Main Program, Elaboration, Validation, Composition

Usage

```
get_instances
```

Returns

A list of child instance names.

Arguments

No arguments.

Example

```
get_instances
```

Notes

This command can be used with instances created by either `add_instance` or `add_hdl_instance`.

Related Information

- add_hdl_instance on page 9-49
- add_instance on page 9-52
- get_instance_parameter_value on page 9-67
- get_instance_parameters on page 9-65
- set_instance_parameter_value on page 9-70

get_instance_interfaces

Description

Returns a list of interfaces found in a child instance. The list of interfaces can change if the parameterization of the instance changes.

Availability

Validation, Composition

Usage

`get_instance_interfaces` *<instance>*

Returns

A list of interface names.

Arguments

instance

The name of the child instance.

Example

```
get_instance_interfaces pixel_converter
```

Related Information

- **add_instance** on page 9-52
- **get_instance_interface_ports** on page 9-59
- **get_instance_interfaces** on page 9-58

get_instance_interface_ports

Description

Returns a list of ports found in an interface of a child instance.

Availability

Validation, Composition, Fileset Generation

Usage

`get_instance_interface_ports` *<instance> <interface>*

Returns

A list of port names found in the interface.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

Example

```
set port_names [ get_instance_interface_ports cpu data_master ]
```

Related Information

- **add_instance** on page 9-52
- **get_instance_interfaces** on page 9-58
- **get_instance_port_property** on page 9-68

get_instance_interface_properties

Description

Returns the names of all of the properties of the specified interface

Availability

Validation, Composition

Usage

`get_instance_interface_properties` *<instance> <interface>*

Returns

List of property names.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the instance.

Example

```
set properties [ get_instance_interface_properties cpu data_master ]
```

Related Information

- **add_instance** on page 9-52
- **get_instance_interface_property** on page 9-64
- **get_instance_interfaces** on page 9-58

get_instance_property

Description

Returns the value of a single instance property.

Availability

Main Program, Elaboration, Validation, Composition, Fileset Generation

Usage

`get_instance_property` *<instance> <property>*

Returns

Various.

Arguments

instance

The name of the instance.

property

The name of the property. Refer to *Instance Properties*.

Example

```
set my_name [ get_instance_property myinstance NAME ]
```

Related Information

- **add_instance** on page 9-52
- **get_instance_properties** on page 9-63
- **set_instance_property** on page 9-62
- **Instance Properties** on page 9-91

set_instance_property

Description

Allows a user to set the properties of a child instance.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`set_instance_property` *<instance> <property> <value>*

Returns

Arguments

instance

The name of the instance.

property

The name of the property to set. Refer to *Instance Properties*.

value

The new property value.

Example

```
set_instance_property myinstance SUPRESS_ALL_WARNINGS true
```

Related Information

- **add_instance** on page 9-52
- **get_instance_properties** on page 9-63
- **get_instance_property** on page 9-61
- **Instance Properties** on page 9-91

get_instance_properties

Description

Returns the names of all the instance properties as a list of strings. You can use the `get_instance_property` and `set_instance_property` commands to get and set values of individual properties. The value returned by this command is always the same for a particular version of Qsys

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_instance_properties`

Returns

List of strings. Refer to *Instance Properties*.

Arguments

No arguments.

Example

```
get_instance_properties
```

Related Information

- **add_instance** on page 9-52
- **get_instance_property** on page 9-61
- **set_instance_property** on page 9-62
- **Instance Properties** on page 9-91

get_instance_interface_property

Description

Returns the value of a property for an interface in a child instance.

Availability

Validation, Composition

Usage

`get_instance_interface_property` *<instance> <interface> <property>*

Returns

The value of the property.

Arguments

instance

The name of the child instance.

interface

The name of an interface on the child instance.

property

The name of the property of the interface.

Example

```
set value [ get_instance_interface_property cpu data_master setupTime ]
```

Related Information

- **add_instance** on page 9-52
- **get_instance_interfaces** on page 9-58

get_instance_parameters

Description

Returns a list of names of the parameters on a child instance that can be set using `set_instance_parameter_value`. It omits parameters that are derived and those that have the `SYSTEM_INFO` parameter property set.

Availability

Main Program, Elaboration, Validation, Composition

Usage

`get_instance_parameters` *<instance>*

Returns

A list of parameters in the instance.

Arguments

instance

The name of the child instance.

Example

```
set parameters [ get_instance_parameters instance ]
```

Notes

You can use this command with instances created by either `add_instance` or `add_hdl_instance`.

Related Information

- **add_hdl_instance** on page 9-49
- **add_instance** on page 9-52
- **get_instance_parameter_value** on page 9-67
- **get_instances** on page 9-57
- **set_instance_parameter_value** on page 9-70

get_instance_parameter_property

Description

Returns the value of a property on a parameter in a child instance. Parameter properties are metadata about how the parameter will be used by the Qsys tools.

Availability

Validation, Composition

Usage

`get_instance_parameter_property` *<instance> <parameter> <property>*

Returns

The value of the parameter property.

Arguments

instance

The name of the child instance.

parameter

The name of the parameter in the instance.

property

The name of the property of the parameter. Refer to *Parameter Properties*.

Example

```
get_instance_parameter_property instance parameter property
```

Related Information

- **add_instance** on page 9-52
- **Parameter Properties** on page 9-92

get_instance_parameter_value

Description

Returns the value of a parameter in a child instance. You cannot use this command to get the value of parameters whose values are derived or those that are defined using the `SYSTEM_INFO` parameter property.

Availability

Elaboration, Validation, Composition

Usage

`get_instance_parameter_value` *<instance> <parameter>*

Returns

The value of the parameter.

Arguments

instance

The name of the child instance.

parameter

Specifies the parameter whose value is being retrieved.

Example

```
set dpi [ get_instance_parameter_value pixel_converter input_DPI ]
```

Notes

You can use this command with instances created by either `add_instance` or `add_hdl_instance`.

Related Information

- **add_hdl_instance** on page 9-49
- **add_instance** on page 9-52
- **get_instance_parameters** on page 9-65
- **get_instances** on page 9-57
- **set_instance_parameter_value** on page 9-70

get_instance_port_property

Description

Returns the value of a property of a port contained by an interface in a child instance.

Availability

Validation, Composition, Fileset Generation

Usage

`get_instance_port_property` *<instance> <port> <property>*

Returns

The value of the property for the port.

Arguments

instance

The name of the child instance.

port

The name of a port in one of the interfaces on the child instance.

property

The property whose value is being retrieved. Only the following port properties can be queried on ports of child instances: `ROLE`, `DIRECTION`, `WIDTH`, `WIDTH_EXPR` and `VHDL_TYPE`. Refer to *Port Properties*.

Example

```
get_instance_port_property instance port property
```

Related Information

- **add_instance** on page 9-52
- **get_instance_interface_ports** on page 9-59
- **Port Properties** on page 9-97

set_connection_parameter_value

Description

Sets the value of a parameter of the connection. The start and end are each interface names of the format `<instance>.<interface>`. Connection parameters depend on the type of connection, for Avalon-MM they include base addresses and arbitration priorities.

Availability

Main Program, Composition

Usage

`set_connection_parameter_value` *<connection> <parameter> <value>*

Returns

No return value.

Arguments

connection

Specifies the name of the connection as returned by the `add_conection`command. It is of the form `start.point/end.point`.

parameter

The name of the parameter.

value

The new parameter value.

Example

```
set_connection_parameter_value cpu.data_master/dma0.csr baseAddress "0x000a0000"
```

Related Information

- add_connection on page 9-52
- get_connection_parameter_value on page 9-56

set_instance_parameter_value

Description

Sets the value of a parameter for a child instance. Derived parameters and `SYSTEM_INFO` parameters for the child instance can not be set with this command.

Availability

Main Program, Elaboration, Composition

Usage

`set_instance_parameter_value` *<instance> <parameter> <value>*

Returns

Vo return value.

Arguments

instance

Specifies the name of the child instance.

parameter

Specifies the parameter that is being set.

value

Specifies the new parameter value.

Example

```
set_instance_parameter_value uart_0 baudRate 9600
```

Notes

You can use this command with instances created by either `add_instance` or `add_hdl_instance`.

Related Information

- add_hdl_instance on page 9-49
- add_instance on page 9-52
- get_instance_parameter_value on page 9-67
- get_instances on page 9-57

Fileset Generation

add_fileset

Description

Adds a generation fileset for a particular target as specified by the `kind`. Qsys calls the target (`SIM_VHDL`, `SIM_VERILOG`, `QUARTUS_SYNTH`, or `EXAMPLE_DESIGN`) when the specified generation target is requested. You can define multiple filesets for each kind of fileset. Qsys passes a single argument to the specified callback procedure. The value of the argument is a generated name, which you must use in the top-level module or entity declaration of your IP component. To override this generated name, you can set the fileset property `TOP_LEVEL`.

Availability

Main Program

Usage

`add_fileset` *<name> <kind>* [*<callback_proc> <display_name>*]

Returns

No return value.

Arguments

name

The name of the fileset.

kind

The kind of fileset. Refer to *Fileset Properties*.

callback_proc (optional)

A string identifying the name of the callback procedure. If you add files in the global section, you can then specify a blank callback procedure.

display_name (optional)

A display string to identify the fileset.

Example

```
add_fileset my_synthesis_fileset QUARTUS_SYNTH mySynthCallbackProc "My Synthesis"
proc mySynthCallbackProc { topLevelName } { ... }
```

Notes

If using the `TOP_LEVEL` fileset property, all parameterizations of the component must use identical HDL.

Related Information

- **add_fileset_file** on page 9-73
- **get_fileset_property** on page 9-78
- **Fileset Properties** on page 9-105

add_fileset_file

Description

Adds a file to the generation directory. You can specify source file locations with either an absolute path, or a path relative to the IP component's `_hw.tcl` file. When you use the `add_fileset_file` command in a fileset callback, the Quartus II software compiles the files in the order that they are added.

Availability

Main Program, Fileset Generation

Usage

`add_fileset_file` *<output_file> <file_type> <file_source> <path_or_contents>* [*<attributes>*]

Returns

No return value.

Arguments

output_file

Specifies the location to store the file after Qsys generation

file_type

The kind of file. Refer to *File Kind Properties*.

file_source

Specifies whether the file is being added by path, or by file contents. Refer to *File Source Properties*.

path_or_contents

When the `file_source` is `PATH`, specifies the file to be copied to `output_file`. When the `file_source` is `TEXT`, specifies the text contents to be stored in the file.

attributes (optional)

An optional list of file attributes. Typically used to specify that a file is intended for use only in a particular simulator. Refer to *File Attribute Properties*.

Example

```
add_fileset_file "./implementation/rx_pma.sv" SYSTEM_VERILOG PATH synth_rx_pma.sv
add_fileset_file gui.sv SYSTEM_VERILOG TEXT "Customize your IP core"
```

Related Information

- **add_fileset** on page 9-72
- **get_fileset_file_attribute** on page 9-75
- **File Kind Properties** on page 9-109
- **File Source Properties** on page 9-110
- **File Attribute Properties** on page 9-108

set_fileset_property

Description

Allows you to set the properties of a fileset.

Availability

Main Program, Elaboration, Fileset Generation

Usage

`set_fileset_property` *<fileset> <property> <value>*

Returns

No return value.

Arguments

fileset

The name of the fileset.

property

The name of the property to set. Refer to *Fileset Properties*.

value

The new property value.

Example

```
set_fileset_property mySynthFileset TOP_LEVEL simple_uart
```

Notes

When a fileset callback is called, the callback procedure will be passed a single argument. The value of this argument is a generated name which must be used in the top-level module or entity declaration of your IP component. If set, the `TOP_LEVEL` specifies a fixed name for the top-level name of your IP component.

The `TOP_LEVEL` property must be set in the global section. It cannot be set in a fileset callback.

If using the `TOP_LEVEL` fileset property, all parameterizations of the IP component must use identical HDL.

Related Information

- **add_fileset** on page 9-72
- **Fileset Properties** on page 9-105

get_fileset_file_attribute

Description

Returns the attribute of a fileset file.

Availability

Main Program, Fileset Generation

Usage

`get_fileset_file_attribute` *<output_file> <attribute>*

Returns

Value of the fileset File attribute.

Arguments

output_file

Location of the output file.

attribute

Specifies the name of the attribute Refer to *File Attribute Properties*.

Example

```
get_fileset_file_attribute my_file.sv ALDEC_SPECIFIC
```

Related Information

- **add_fileset** on page 9-72
- **add_fileset_file** on page 9-73
- **get_fileset_file_attribute** on page 9-75
- **File Attribute Properties** on page 9-108
- **add_fileset** on page 9-72
- **add_fileset_file** on page 9-73
- **get_fileset_file_attribute** on page 9-75
- **File Attribute Properties** on page 9-108

set_fileset_file_attribute

Description

Sets the attribute of a fileset file.

Availability

Main Program, Fileset Generation

Usage

`set_fileset_file_attribute` *<output_file> <attribute> <value>*

Returns

The attribute value if it was set.

Arguments

output_file

Location of the output file.

attribute

Specifies the name of the attribute Refer to *File Attribute Properties*.

value

Value to set the attribute to.

Example

```
set_fileset_file_attribute my_file_pkg.sv COMMON_SYSTEMVERILOG_PACKAGE
my_file_package
```

get_fileset_properties

Description

Returns a list of properties that can be set on a fileset.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

Usage

```
get_fileset_properties
```

Returns

A list of property names. Refer to *Fileset Properties*.

Arguments

No arguments.

Example

```
get_fileset_properties
```

Related Information

- add_fileset on page 9-72
- get_fileset_properties on page 9-77
- set_fileset_property on page 9-74
- Fileset Properties on page 9-105

get_fileset_property

Description

Returns the value of a fileset property for a fileset.

Availability

Main Program, Elaboration, Fileset Generation

Usage

`get_fileset_property` *<fileset> <property>*

Returns

The value of the property.

Arguments

fileset

The name of the fileset.

property

The name of the property to query. Refer to *Fileset Properties*.

Example

```
get_fileset_property fileset property
```

Related Information

Fileset Properties on page 9-105

get_fileset_sim_properties

Description

Returns simulator properties for a fileset.

Availability

Main Program, Fileset Generation

Usage

`get_fileset_sim_properties` *<fileset> <platform> <property>*

Returns

The fileset simulator properties.

Arguments

fileset

The name of the fileset.

platform

The operating system for that applies to the property. Refer to *Operating System Properties*.

property

Specifies the name of the property to set. Refer to *Simulator Properties*.

Example

```
get_fileset_sim_properties my_fileset LINUX64 OPT_CADENCE_64BIT
```

Related Information

- **add_fileset** on page 9-72
- **set_fileset_sim_properties** on page 9-80
- **Operating System Properties** on page 9-117
- **Simulator Properties** on page 9-111

set_fileset_sim_properties

Description

Sets simulator properties for a given fileset

Availability

Main Program, Fileset Generation

Usage

`set_fileset_sim_properties` *<fileset> <platform> <property> <value>*

Returns

The fileset simulator properties if they were set.

Arguments

fileset

The name of the fileset.

platform

The operating system that applies to the property. Refer to *Operating System Properties*.

property

Specifies the name of the property to set. Refer to *Simulator Properties*.

value

Specifies the value of the property.

Example

```
set_fileset_sim_properties my_fileset LINUX64 OPT_MENTOR_PLI "{libA} {libB}"
```

Related Information

- **get_fileset_sim_properties** on page 9-79
- **Operating System Properties** on page 9-117
- **Simulator Properties** on page 9-111

create_temp_file

Description

Creates a temporary file, which you can use inside the fileset callbacks of a **_hw.tcl** file. This temporary file is included in the generation output if it is added using the `add_fileset_file` command.

Availability

Fileset Generation

Usage

`create_temp_file` <*path*>

Returns

The path to the temporary file.

Arguments

path

The name of the temporary file.

Example

```
set filelocation [create_temp_file "./hdl/compute_frequency.v" ]
add_fileset_file compute_frequency.v VERILOG PATH ${filelocation}
```

Related Information

- **add_fileset** on page 9-72
- **add_fileset_file** on page 9-73

Miscellaneous

check_device_family_equivalence

Description

Returns 1 if the device family is equivalent to one of the families in the device families lis., Returns 0 if the device family is not equivalent to any families. This command ignores differences in capitalization and spaces.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Composition, Fileset Generation, Parameter Upgrade

Usage

`check_device_family_equivalence` *<device_family> <device_family_list>*

Returns

1 if equivalent, 0 if not equivalent.

Arguments

device_family

The device family name that is being checked.

device_family_list

The list of device family names to check against.

Example

```
check_device_family_equivalence "CYLCONE III LS" { "stratixv" "Cyclone IV"
"cycloneiiils" }
```

Related Information

get_device_family_displayname on page 9-84

get_device_family_displayname

Description

Returns the display name of a given device family.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Composition, Fileset Generation, Parameter Upgrade

Usage

`get_device_family_displayname` *<device_family>*

Returns

The preferred display name for the device family.

Arguments

device_family

A device family name.

Example

```
get_device_family_displayname cycloneiiils ( returns: "Cyclone IV LS" )
```

Related Information

check_device_family_equivalence on page 9-83

get_qip_strings

Description

Returns a Tcl list of QIP strings for the IP component.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Composition, Parameter Upgrade

Usage

```
get_qip_strings
```

Returns

A Tcl list of qip strings set by this IP component.

Arguments

No arguments.

Example

```
set strings [ get_qip_strings ]
```

Related Information

set_qip_strings on page 9-86

set_qip_strings

Description

Places strings in the Quartus II IP File (**.qip**) file, which Qsys passes to the command as a Tcl list. You add the **.qip** file to your Quartus II project on the **Files** page, in the **Settings** dialog box. Successive calls to `set_qip_strings` are not additive and replace the previously declared value.

Availability

Discovery, Main Program, Edit, Elaboration, Validation, Composition, Parameter Upgrade

Usage

`set_qip_strings` *<qip_strings>*

Returns

The Tcl list which was set.

Arguments

qip_strings

A space-delimited Tcl list.

Example

```
set_qip_strings {"QIP Entry 1" "QIP Entry 2"}
```

Notes

You can use the following macros in your QIP strings entry:

`%entityName%`	The generated name of the entity replaces this macro when the string is written to the **.qip** file.
`%libraryName%`	The compilation library this IP component was compiled into is inserted in place of this macro inside the **.qip** file.
`%instanceName%`	The name of the instance is inserted in place of this macro inside the **.qip** file.

Related Information

get_qip_strings on page 9-85

set_interconnect_requirement

Description

Sets the value of an interconnect requirement for a system or an interface on a child instance.

Availability

Composition

Usage

`set_interconnect_requirement` *<element_id> <name> <value>*

Returns

No return value

Arguments

element_id

`{$system}` for system requirements, or qualified name of the interface of an instance, in `<instance>.<interface>` format. Note that the system identifier has to be escaped in TCL.

name

The name of the requirement.

value

The new requirement value.

Example

```
set_interconnect_requirement {$system} qsys_mm.maxAdditionalLatency 2
```

Qsys _hw.tcl Property Reference

Script Language Properties

Name	Description
`TCL`	Implements the script in Tcl.

Interface Properties

Name	Description
`CMSIS_SVD_FILE`	Specifies the connection point's associated CMSIS file.
`CMSIS_SVD_VARIABLES`	Defines the variables inside a .svd file.
`ENABLED`	Specifies whether or not interface is enabled.
`EXPORT_OF`	For composed **_hwl.tcl** files, the `EXPORT_OF` property indicates which interface of a child instance is to be exported through this interface. Before using this command, you must have created the border interface using `add_interface`. The interface to be exported is of the form <*`instanceName.interfaceName`*>. Example: `set_interface_property CSC_input EXPORT_OF my_colorSpace-Converter.input_port`
`PORT_NAME_MAP`	A map of external port names to internal port names, formatted as a Tcl list. Example: `set_interface_property <interface name> PORT_NAME_MAP "<new port name> <old port name> <new port name 2> <old port name 2>"`
`SVD_ADDRESS_GROUP`	Generates a CMSIS SVD file. Masters in the same SVD address group will write register data of their connected slaves into the same SVD file
`SVD_ADDRESS_OFFSET`	Generates a CMSIS SVD file. Slaves connected to this master will have their base address offset by this amount in the SVD file.

Instance Properties

Name	Description
`HDLINSTANCE_GET_GENERATED_NAME`	Qsys uses this property to get the auto-generated fixed name when the instance property `HDLINSTANCE_USE_GENERATED_NAME` is set to true, and only applies to fileSet callbacks.
`HDLINSTANCE_USE_GENERATED_NAME`	If true, instances added with the `add_hdl_instance` command are instructed to use unique auto-generated fixed names based on the parameterization.
`SUPPRESS_ALL_INFO_MESSAGES`	If true, allows you to suppress all Info messages that originate from a child instance.
`SUPPRESS_ALL_WARNINGS`	If true, allows you to suppress alL warnings that originate from a child instance

Parameter Properties

Type	Name	Description
Boolean	`AFFECTS_ELABORATION`	Set `AFFECTS_ELABORATION` to `false` for parameters that do not affect the external interface of the module. An example of a parameter that does not affect the external interface is `isNonVolatileStorage`. An example of a parameter that does affect the external interface is `width`. When the value of a parameter changes, if that parameter has set `AFFECTS_ELABORATION=false`, the elaboration phase (calling the callback or hardware analysis) is not repeated, improving performance. Because the default value of `AFFECTS_ELABORATION` is `true`, the provided HDL file is normally re-analyzed to determine the new port widths and configuration every time a parameter changes.
Boolean	`AFFECTS_GENERATION`	The default value of `AFFECTS_GENERATION` is `false` if you provide a top-level HDL module; it is `true` if you provide a fileset callback. Set `AFFECTS_GENERATION` to false if the value of a parameter does not change the results of fileset generation.
Boolean	`AFFECTS_VALIDATION`	The `AFFECTS_VALIDATION` property marks whether a parameter's value is used to set derived parameters, and whether the value affects validation messages. When set to `false`, this may improve response time in the parameter editor UI when the value is changed.
String[]	`ALLOWED_RANGES`	Indicates the range or ranges that the parameter value can have. For integers, The `ALLOWED_RANGES` property is a list of ranges that the parameter can take on, where each range is a single value, or a range of values defined by a start and end value separated by a colon, such as `11:15`. This property can also specify legal values and display strings for integers, such as `{0:None 1:Monophonic 2:Stereo 4:Quadrophonic}` meaning 0, 1, 2, and 4 are the legal values. You can also assign display strings to be displayed in the parameter editor for string variables. For example, `ALLOWED_RANGES {"dev1:Cyclone IV GX""dev2:Stratix V GT"}`.
String	`DEFAULT_VALUE`	The default value.
Boolean	`DERIVED`	When `true`, indicates that the parameter value can only be set by the IP component, and cannot be set by the user. Derived parameters are not saved as part of an instance's parameter values. The default value is `false`.
String	`DESCRIPTION`	A short user-visible description of the parameter, suitable for a tooltip description in the parameter editor.
String[]	`DISPLAY_HINT`	Provides a hint about how to display a property. The following values are possible:

Type	Name	Description
		• `boolean`--for `integer` parameters whose value can be 0 or 1. The parameter displays as an option that you can turn on or off. • `radio`--displays a parameter with a list of values as radio buttons instead of a drop-down list. • `hexadecimal`--for `integer` parameters, display and interpret the value as a hexadecimal number, for example: `0x00000010` instead of `16`. • `fixed_size`--for `string_list` and `integer_list` parameters, the `fixed_size DISPLAY_HINT` eliminates the **add** and **remove** buttons from tables.
String	`DISPLAY_NAME`	This is the GUI label that appears to the left of this parameter.
String	`DISPLAY_UNITS`	This is the GUI label that appears to the right of the parameter.
Boolean	`ENABLED`	When `false`, the parameter is disabled, meaning that it is displayed, but greyed out, indicating that it is not editable on the parameter editor.
String	`GROUP`	Controls the layout of parameters in GUI
Boolean	`HDL_PARAMETER`	When true, the parameter must be passed to the HDL IP component description. The default value is `false`.
String	`LONG_DESCRIPTION`	A user-visible description of the parameter. Similar to `DESCRIPTION`, but allows for a more detailed explanation.
String	`NEW_INSTANCE_VALUE`	This property allows you to change the default value of a parameter without affecting older IP components that have did not explicitly set a parameter value, and use the `DEFAULT_VALUE` property. The practical result is that older instances will continue to use `DEFAULT_VALUE` for the parameter and new instances will use the value assigned by `NEW_INSTANCE_VALUE`.
String[]	`SYSTEM_INFO`	Allows you to assign information about the instantiating system to a parameter that you define. `SYSTEM_INFO` requires an argument specifying the type of information requested, <*`info-type`*>.
String	`SYSTEM_INFO_ARG`	Defines an argument to be passed to a particular `SYSTEM_INFO` function, such as the name of a reset interface.
(various)	`SYSTEM_INFO_TYPE`	Specifies one of the types of system information that can be queried. Refer to *System Info Type Properties*.
(various)	`TYPE`	Specifies the type of the parameter. Refer to *Parameter Type Properties*.
(various)	`UNITS`	Sets the units of the parameter. Refer to *Units Properties*.
Boolean	`VISIBLE`	Indicates whether or not to display the parameter in the parameterization GUI.
String	`WIDTH`	For a `STD_LOGIC_VECTOR` parameter, this indicates the width of the logic vector.

Related Information

- **System Info Type Properties** on page 9-113
- **Parameter Type Properties** on page 9-95
- **Units Properties** on page 9-116

Parameter Type Properties

Name	Description
BOOLEAN	A boolean parameter whose value is `true` or `false`.
FLOAT	A signed 32-bit floating point parameter. Not supported for HDL parameters.
INTEGER	A signed 32-bit integer parameter.
INTEGER_LIST	A parameter that contains a list of 32-bit integers. Not supported for HDL parameters.
LONG	A signed 64-bit integer parameter. Not supported for HDL parameters.
NATURAL	A 32-bit number that contain values `0` to `2147483647` (`0x7fffffff`).
POSITIVE	A 32-bit number that contains values `1` to `2147483647` (`0x7fffffff`).
STD_LOGIC	A single bit parameter whose value can be `1` or `0`;
STD_LOGIC_VECTOR	An arbitrary-width number. The parameter property `WIDTH` determines the size of the logic vector.
STRING	A string parameter.
STRING_LIST	A parameter that contains a list of strings. Not supported for HDL parameters.

Parameter Status Properties

Type	Name	Description
Boolean	`ACTIVE`	Indicates the parameter is a regular parameter.
Boolean	`DEPRECATED`	Indicates the parameter exists only for backwards compatibility, and may not have any effect.
Boolean	`EXPERIMENTAL`	Indicates the parameter is experimental, and not exposed in the design flow.

Port Properties

Type	Name	Description
(various)	`DIRECTION`	The direction of the port from the IP component's perspective. Refer to *Direction Properties*.
String	`DRIVEN_BY`	Indicates that this output port is always driven to a constant value or by an input port. If all outputs on an IP component specify a `driven_by` property, the HDL for the IP component will be generated automatically.
String[]	`FRAGMENT_LIST`	This property can be used in 2 ways: First you can take a single RTL signal and split it into multiple Qsys signals `add_interface_port <interface> foo <role> <direction> <width> add_interface_port <interface> bar <role> <direction> <width> set_port_property foo fragment_list "my_rtl_signal(3:0)" set_port_property bar fragment_list "my_rtl_signal(6:4)"` Second you can take multiple RTL signals and combine them into a single Qsys signal `add_interface_port <interface> baz <role> <direction> <width> set_port_property baz fragment_list "rtl_signal_1(3:0) rtl_signal_2(3:0)"` Note: The listed bits in a port fragment must match the declared width of the Qsys signal.
String	`ROLE`	Specifies an Avalon signal type such as `waitrequest`, `readdata`, or `read`. For a complete list of signal types, refer to the *Avalon Interface Specifications*.
Boolean	`TERMINATION`	When `true`, instead of connecting the port to the Qsys system, it is left unconnected for `output` and `bidir` or set to a fixed value for `input`. Has no effect for IP components that implement a generation callback instead of using the default wrapper generation.
BigInteger	`TERMINATION_VALUE`	The constant value to drive an input port.
(various)	`VHDL_TYPE`	Indicates the type of a VHDL port. The default value, `auto`, selects `std_logic` if the width is fixed at 1, and `std_logic_vector` otherwise. Refer to *Port VHDL Type Properties*.
String	`WIDTH`	The width of the port in bits. Cannot be set directly. Any changes must be set through the `WIDTH_EXPR` property.
String	`WIDTH_EXPR`	The width expression of a port. The `width_value_expr` property can be set directly to a numeric value if desired. When `get_port_property` is used width always returns the current integer width of the port while `width_expr` always returns the unevaluated width expression.
Integer	`WIDTH_VALUE`	The width of the port in bits. Cannot be set directly. Any changes must be set through the `WIDTH_EXPR` property.

Related Information

- **Direction Properties** on page 9-99

- **Port VHDL Type Properties** on page 9-112
- **Avalon Interface Specifications**

Direction Properties

Name	Description
`Bidir`	Direction for a bidirectional signal.
`Input`	Direction for an input signal.
`Output`	Direction for an output signal.

Display Item Properties

Type	Name	Description
String	`DESCRIPTION`	A description of the display item, which you can use as a tooltip.
String[]	`DISPLAY_HINT`	A hint that affects how the display item displays in the parameter editor.
String	`DISPLAY_NAME`	The label for the display item in a the parameter editor.
Boolean	`ENABLED`	Indicates whether the display item is enabled or disabled.
String	`PATH`	The path to a file. Only applies to display items of type `ICON`.
String	`TEXT`	Text associated with a display item. Only applies to display items of type `TEXT`.
Boolean	`VISIBLE`	Indicates whether this display item is visible or not.

Display Item Kind Properties

Name	Description
`ACTION`	An action displays as a button in the GUI. When the button is clicked, it calls the callback procedure. The button label is the display item `id`.
`GROUP`	A group that is a child of the `parent_group` group. If the `parent_group` is an empty string, this is a top-level group.
`ICON`	A **.gif**, **.jpg**, or **.png** file.
`PARAMETER`	A parameter in the instance.
`TEXT`	A block of text.

Display Hint Properties

Name	Description
`BIT_WIDTH`	Bit width of a number
`BOOLEAN`	Integer value either `0` or `1`.
`COLLAPSED`	Indicates whether a group is collapsed when initially displayed.
`COLUMNS`	Number of columns in text field, for example, `"columns:N"`.
`EDITABLE`	Indicates whether a list of strings allows free-form text entry (editable combo box).
`FILE`	Indicates that the string is an optional file path, for example, **"file:jpg,png,gif"**.
`FIXED_SIZE`	Indicates a fixed size for a table or list.
`GROW`	if set, the widget can grow when the IP component is resized.
`HEXADECIMAL`	Indicates that the long integer is hexadecimal.
`RADIO`	Indicates that the range displays as radio buttons.
`ROWS`	Number of rows in text field, or visible rows in a table, for example, `"rows:N"`.
`SLIDER`	Range displays as slider.
`TAB`	if present for a group, the group displays in a tab
`TABLE`	if present for a group, the group must contain all list-type parameters, which display collectively in a single table.
`TEXT`	String is a text field with a limited character set, for example, `"text:A-Za-z0-9_"`.
`WIDTH`	width of a table column

Module Properties

Name	Description
ANALYZE_HDL	When set to false, prevents a call to the Quartus II mapper to verify port widths and directions, speeding up generation time at the expense of fewer validation checks. If this property is set to false, invalid port widths and directions are discovered during the Quartus II software compilation. This does not affect IP components using filesets to manage synthesis files.
AUTHOR	The IP component author.
COMPOSITION_CALLBACK	The name of the composition callback. If you define a composition callback, you cannot not define the generation or elaboration callbacks.
DATASHEET_URL	Deprecated. Use `add_documentation_link` to provide documentation links.
DESCRIPTION	The description of the IP component, such as "This IP component puts the shizzle in the frobnitz."
DISPLAY_NAME	The name to display when referencing the IP component, such as "My Qsys IP Component".
EDITABLE	Indicates whether you can edit the IP component in the Component Editor.
ELABORATION_CALLBACK	The name of the elaboration callback. When set, the IP component's elaboration callback is called to validate and elaborate interfaces for instances of the IP component.
GENERATION_CALLBACK	The name for a custom generation callback.
GROUP	The group in the IP Catalog that includes this IP component.
ICON_PATH	A path to an icon to display in the IP component's parameter editor.
INSTANTIATE_IN_SYSTEM_MODULE	If true, this IP component is implemented by HDL provided by the IP component. If false, the IP component will create exported interfaces allowing the implementation to be connected in the parent.
INTERNAL	An IP component which is marked as internal does not appear in the IP Catalog. This feature allows you to hide the sub-IP-components of a larger composed IP component.
MODULE_DIRECTORY	The directory in which the hw.tcl file exists.
MODULE_TCL_FILE	The path to the hw.tcl file.
NAME	The name of the IP component, such as `my_qsys_component`.
OPAQUE_ADDRESS_MAP	For composed IP components created using a _hw.tcl file that include children that are memory-mapped slaves, specifies whether the children's addresses are visible to downstream

Name	Description
	software tools. When `true`, the children's address are not visible. When `false`, the children's addresses are visible.
`PREFERRED_SIMULATION_LANGUAGE`	The preferred language to use for selecting the fileset for simulation model generation.
`REPORT_HIERARCHY`	null
`STATIC_TOP_LEVEL_MODULE_NAME`	Deprecated.
`STRUCTURAL_COMPOSITION_CALLBACK`	The name of the structural composition callback. This callback is used to represent the structural hierarchical model of the IP component and the RTL can be generated either with module property `COMPOSITION_CALLBACK` or by `ADD_FILESET` with target `QUARTUS_SYNTH`
`SUPPORTED_DEVICE_FAMILIES`	A list of device family supported by this IP component.
`TOP_LEVEL_HDL_FILE`	Deprecated.
`TOP_LEVEL_HDL_MODULE`	Deprecated.
`UPGRADEABLE_FROM`	null
`VALIDATION_CALLBACK`	The name of the validation callback procedure.
`VERSION`	The IP component's version, such as 10.0.

Fileset Properties

Name	Description
`ENABLE_FILE_OVERWRITE_MODE`	null
`ENABLE_RELATIVE_INCLUDE_PATHS`	If true, HDL files can include other files using relative paths in the fileset.
`TOP_LEVEL`	The name of the top-level HDL module that the fileset generates. If set, the HDL top level must match the `TOP_LEVEL` name, and the HDL must not be parameterized. Qsys runs the generate callback one time, regardless of the number of instances in the system.

Fileset Kind Properties

Name	Description
EXAMPLE_DESIGN	Contains example design files.
QUARTUS_SYNTH	Contains files that Qsys uses for the Quartus II software synthesis.
SIM_VERILOG	Contains files that Qsys uses for Verilog HDL simulation.
SIM_VHDL	Contains files that Qsys uses for VHDL simulation.

Callback Properties

Description

This list describes each type of callback. Each command may only be available in some callback contexts.

Name	Description
`ACTION`	Called when an `ACTION` display item's action is performed.
`COMPOSITION`	Called during instance elaboration when the IP component contains a subsystem.
`EDITOR`	Called when the IP component is controlling the parameterization editor.
`ELABORATION`	Called to elaborate interfaces and signals after a parameter change. In API 9.1 and later, validation is called before elaboration. In API 9.0 and earlier, elaboration is called before validation.
`GENERATE_VERILOG_SIMULATION`	Called when the IP component uses a custom generator to generates the Verilog simulation model for an instance.
`GENERATE_VHDL_SIMULATION`	Called when the IP component uses a custom generator to generates the VHDL simulation model for an instance.
`GENERATION`	Called when the IP component uses a custom generator to generates the synthesis HDL for an instance.
`PARAMETER_UPGRADE`	Called when attempting to instantiate an IP component with a newer version than the saved version. This allows the IP component to upgrade parameters between released versions of the component.
`STRUCTURAL_COMPOSITION`	Called during instance elaboration when an IP component is represented by a structural hierarchical model which may be different from the generated RTL.
`VALIDATION`	Called to validate parameter ranges and report problems with the parameter values. In API 9.1 and later, validation is called before elaboration. In API 9.0 and earlier, elaboration is called before validation.

File Attribute Properties

Name	Description
`ALDEC_SPECIFIC`	Applies to Aldec simulation tools and for simulation filesets only.
`CADENCE_SPECIFIC`	Applies to Cadence simulation tools and for simulation filesets only.
`COMMON_SYSTEMVERILOG_PACKAGE`	The name of the common SystemVerilog package. Applies to simulation filesets only.
`MENTOR_SPECIFIC`	Applies to Mentor simulation tools and for simulation filesets only.
`SYNOPSYS_SPECIFIC`	Applies to Synopsys simulation tools and for simulation filesets only.
`TOP_LEVEL_FILE`	Contains the top-level module for the fileset and applies to synthesis filesets only.

File Kind Properties

Name	Description
DAT	DAT Data
FLI_LIBRARY	FLI Library
HEX	HEX Data
MIF	MIF Data
OTHER	Other
PLI_LIBRARY	PLI Library
QXP	QXP File
SDC	Timing Constraints
SYSTEM_VERILOG	System Verilog HDL
SYSTEM_VERILOG_ENCRYPT	Encrypted System Verilog HDL
SYSTEM_VERILOG_INCLUDE	System Verilog Include
VERILOG	Verilog HDL
VERILOG_ENCRYPT	Encrypted Verilog HDL
VERILOG_INCLUDE	Verilog Include
VHDL	VHDL
VHDL_ENCRYPT	Encrypted VHDL
VPI_LIBRARY	VPI Library

File Source Properties

Name	Description
`PATH`	Specifies the original source file and copies to `output_file`.
`TEXT`	Specifies an arbitrary text string for the contents of `output_file`.

Simulator Properties

Name	Description
ENV_ALDEC_LD_LIBRARY_PATH	LD_LIBRARY_PATH when running riviera-pro
ENV_CADENCE_LD_LIBRARY_PATH	LD_LIBRARY_PATH when running ncsim
ENV_MENTOR_LD_LIBRARY_PATH	LD_LIBRARY_PATH when running modelsim
ENV_SYNOPSYS_LD_LIBRARY_PATH	LD_LIBRARY_PATH when running vcs
OPT_ALDEC_PLI	-pli option for riviera-pro
OPT_CADENCE_64BIT	-64bit option for ncsim
OPT_CADENCE_PLI	-loadpli1 option for ncsim
OPT_CADENCE_SVLIB	-sv_lib option for ncsim
OPT_CADENCE_SVROOT	-sv_root option for ncsim
OPT_MENTOR_64	-64 option for modelsim
OPT_MENTOR_CPPPATH	-cpppath option for modelsim
OPT_MENTOR_LDFLAGS	-ldflags option for modelsim
OPT_MENTOR_PLI	-pli option for modelsim
OPT_SYNOPSYS_ACC	+acc option for vcs
OPT_SYNOPSYS_CPP	-cpp option for vcs
OPT_SYNOPSYS_FULL64	-full64 option for vcs
OPT_SYNOPSYS_LDFLAGS	-LDFLAGS option for vcs
OPT_SYNOPSYS_LLIB	-l option for vcs
OPT_SYNOPSYS_VPI	+vpi option for vcs

Port VHDL Type Properties

Name	Description
`AUTO`	The VHDL type of this signal is automatically determined. Single-bit signals are `STD_LOGIC`; signals wider than one bit will be `STD_LOGIC_VECTOR`.
`STD_LOGIC`	Indicates that the signal in not rendered in VHDL as a `STD_LOGIC` signal.
`STD_LOGIC_VECTOR`	Indicates that the signal is rendered in VHDL as a `STD_LOGIC_VECTOR` signal.

System Info Type Properties

Type	Name	Description
String	`ADDRESS_MAP`	An XML-formatted string describing the address map for the interface specified in the system info argument.
Integer	`ADDRESS_WIDTH`	The number of address bits required to address all memory-mapped slaves connected to the specified memory-mapped master in this instance, using byte addresses.
String	`AVALON_SPEC`	The version of the interconnect. SOPC Builder interconnect uses Avalon Specification 1.0. Qsys interconnect uses Avalon Specification 2.0.
Integer	`CLOCK_DOMAIN`	An integer that represents the clock domain for the interface specified in the system info argument. If this instance has interfaces on multiple clock domains, this can be used to determine which interfaces are on each clock domain. The absolute value of the integer is arbitrary.
Long, Integer	`CLOCK_RATE`	The rate of the clock connected to the clock input specified in the system info argument. If 0, the clock rate is currently unknown.
String	`CLOCK_RESET_INFO`	The name of this instance's primary clock or reset sink interface. This is used to determine the reset sink to use for global reset when using SOPC interconnect.
String	`CUSTOM_INSTRUCTION_SLAVES`	Provides custom instruction slave information, including the name, base address, address span, and clock cycle type.
(various)	`DESIGN_ENVIRONMENT`	A string that identifies the current design environment. Refer to *Design Environment Type Properties*.
String	`DEVICE`	The device part number of the currently selected device.
String	`DEVICE_FAMILY`	The family name of the currently selected device.
String	`DEVICE_FEATURES`	A list of key/value pairs delineated by spaces indicating whether a particular device feature is available in the currently selected device family. The format of the list is suitable for passing to the Tcl `array` set command. The keys are device features; the values will be 1 if the feature is present, and 0 if the feature is absent.
String	`DEVICE_SPEEDGRADE`	The speed grade of the currently selected device.
Integer	`GENERATION_ID`	A integer that stores a hash of the generation time to be used as a unique ID for a generation run.

Type	Name	Description
BigInteger, Long	`INTERRUPTS_USED`	A mask indicating which bits of an interrupt receiver are connected to interrupt senders. The interrupt receiver is specified in the system info argument.
Integer	`MAX_SLAVE_DATA_WIDTH`	The data width of the widest slave connected to the specified memory-mapped master.
String, Boolean, Integer	`QUARTUS_INI`	The value of the quartus.ini setting specified in the system info argument.
Integer	`RESET_DOMAIN`	An integer that represents the reset domain for the interface specified in the system info argument. If this instance has interfaces on multiple reset domains, this can be used to determine which interfaces are on each reset domain. The absolute value of the integer is arbitrary.
String	`TRISTATECONDUIT_INFO`	An XML description of the Avalon Tri-state Conduit masters connected to an Avalon Tri-state Conduit slave. The slave is specified as the system info argument. The value will contain information about the slave, connected master instance and interface names, and signal names, directions and widths.
String	`TRISTATECONDUIT_MASTERS`	The names of the instance's interfaces that are tri-state conduit slaves.
String	`UNIQUE_ID`	A string guaranteed to be unique to this instance.

Related Information

Design Environment Type Properties on page 9-115

Design Environment Type Properties

Description

A design environment is used by IP to tell what sort of interfaces are most appropriate to connect in the parent system.

Name	Description
`NATIVE`	Design environment prefers native IP interfaces.
`QSYS`	Design environment prefers standard Qsys interfaces.

Units Properties

Name	Description
`Address`	A memory-mapped address.
`Bits`	Memory size, in bits.
`BitsPerSecond`	Rate, in bits per second.
`Bytes`	Memory size, in bytes.
`Cycles`	A latency or count, in clock cycles.
`GigabitsPerSecond`	Rate, in gigabits per second.
`Gigabytes`	Memory size, in gigabytes.
`Gigahertz`	Frequency, in GHz.
`Hertz`	Frequency, in Hz.
`KilobitsPerSecond`	Rate, in kilobits per second.
`Kilobytes`	Memory size, in kilobytes.
`Kilohertz`	Frequency, in kHz.
`MegabitsPerSecond`	Rate, in megabits per second.
`Megabytes`	Memory size, in megabytes.
`Megahertz`	Frequency, in MHz.
`Microseconds`	Time, in micros.
`Milliseconds`	Time, in ms.
`Nanoseconds`	Time, in ns.
`None`	Unspecified units.
`Percent`	A percentage.
`Picoseconds`	Time, in ps.
`Seconds`	Time, in s.

Operating System Properties

Name	Description
`ALL`	All operating systems
`LINUX32`	Linux 32-bit
`LINUX64`	Linux 64-bit
`WINDOWS32`	Windows 32-bit
`WINDOWS64`	Windows 64-bit

Quartus.ini Type Properties

Name	Description
`ENABLED`	Returns `1` if the setting is turned on, otherwise returns `0`.
`STRING`	Returns the string value of the **.ini** setting.

Document Revision History

The table below indicates edits made to the *Component Interface Tcl Reference* content since its creation.

Date	Version	Changes
2015.05.04	15.0.0	Edit to `add_fileset_file` command.
December 2014	14.1.0	• set_interface_upgrade_map • Moved **Port Roles (Interface Signal Types)** section to *Qsys Interconnect*.
November 2013	13.1.0	• add_hdl_instance
May 2013	13.0.0	• Consolidated content from other Qsys chapters. • Added AMBA APB support.
November 2012	12.1.0	• Added the **demo_axi_memory** example with screen shots and example **_hw.tcl** code.
June 2012	12.0.0	• Added AMBA AXI3 support. • Added: `set_display_item_property`, `set_parameter_property`,`LONG_DESCRIPTION`, and static filesets.
November 2011	11.1.0	• Template update. • Added: `set_qip_strings`, `get_qip_strings`, `get_device_family_displayname`, `check_device_family_equivalence`.
May 2011	11.0.0	• Revised section describing HDL and composed component implementations. • Separated reset and clock interfaces in example. • Added: `TRISTATECONDUIT_INFO`, `GENERATION_ID`, `UNIQUE_ID` `SYSTEM_INFO`. • Added: `WIDTH` and `SYSTEM_INFO_ARG` parameter properties. • Removed the `doc_type` argument from the `add_documentation_link` command. • Removed: `get_instance_parameter_properties` • Added: `add_fileset`, `add_fileset_file`, `create_temp_file`. • Updated Tcl examples to show separate clock and reset interfaces.
December 2010	10.1.0	• Initial release.

Related Information

Quartus II Handbook Archive

Qsys Interconnect on page 7-1

Qsys System Design Components 10

2015.05.04

QII5V1 Subscribe Send Feedback

You can use Qsys IP components to create Qsys systems. Qsys interfaces include components appropriate for streaming high-speed data, reading and writing registers and memory, controlling off-chip devices, and transporting data between components.

Qsys supports Avalon, AMBA AXI3 (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB3 (version 1.0) interface specifications.

Related Information

- **Avalon Interface Specifications**
- **AMBA Protocol Specifications**
- **Creating a System with Qsys** on page 5-1
- **Qsys Interconnect** on page 7-1
- **Embedded Peripherals IP User Guide**

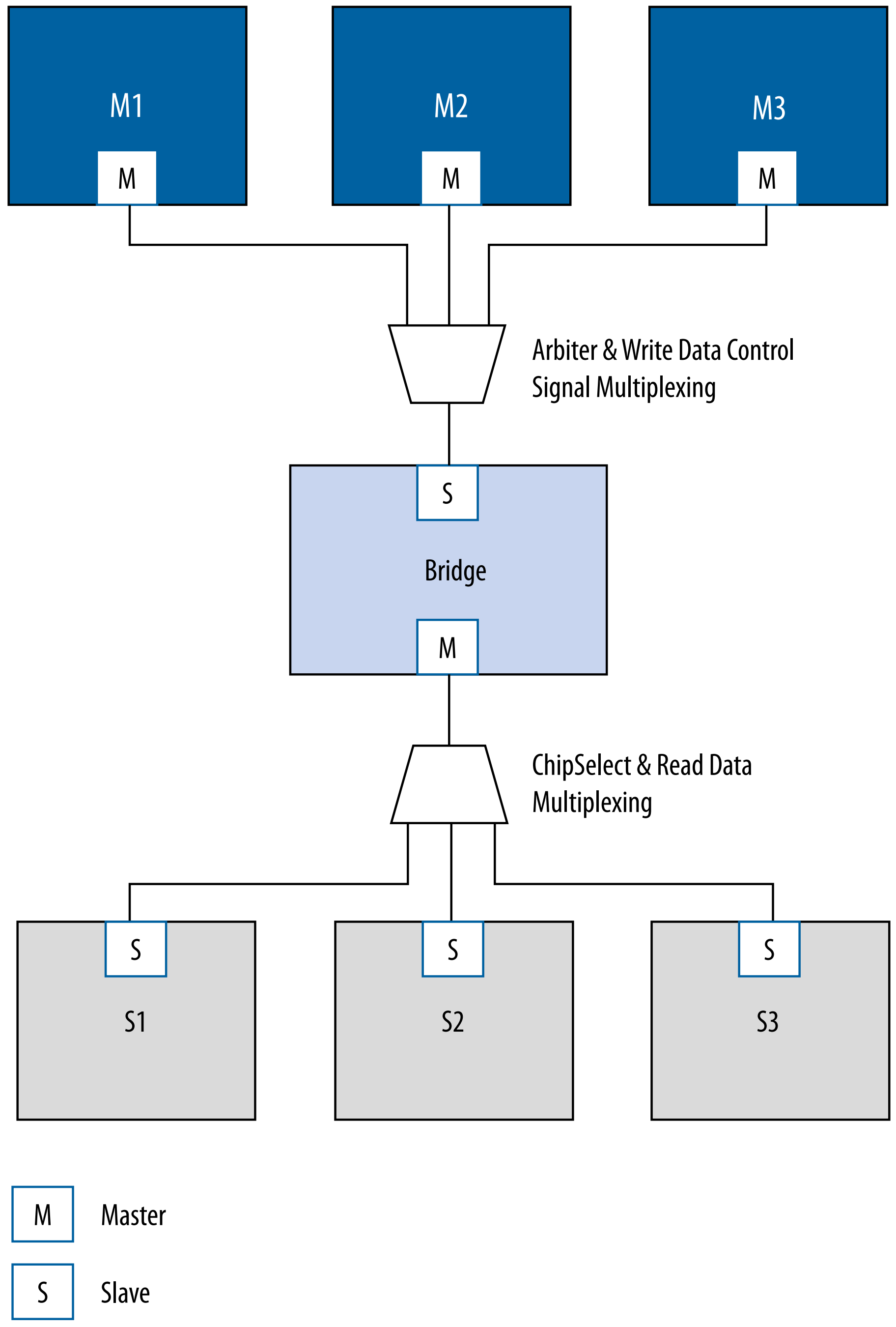
Bridges

Bridges affect the way Qsys transports data between components. You can insert bridges between masters and slave interfaces to control the topology of a Qsys system, which affects the interconnect that Qsys generates. You can also use bridges to separate components into different clock domains to isolate clock domain crossing logic.

A bridge has one slave interface and one master interface. In Qsys, one or more master interfaces from other components connect to the bridge slave. The bridge master connects to one or more slave interfaces on other components.

Figure 10-1: Using a Bridge in a Qsys System

In this example, three masters have logical connections to three slaves, although physically each master connects only to the bridge. Transfers initiated to the slave propagate to the master in the same order in which they are initiated on the slave.

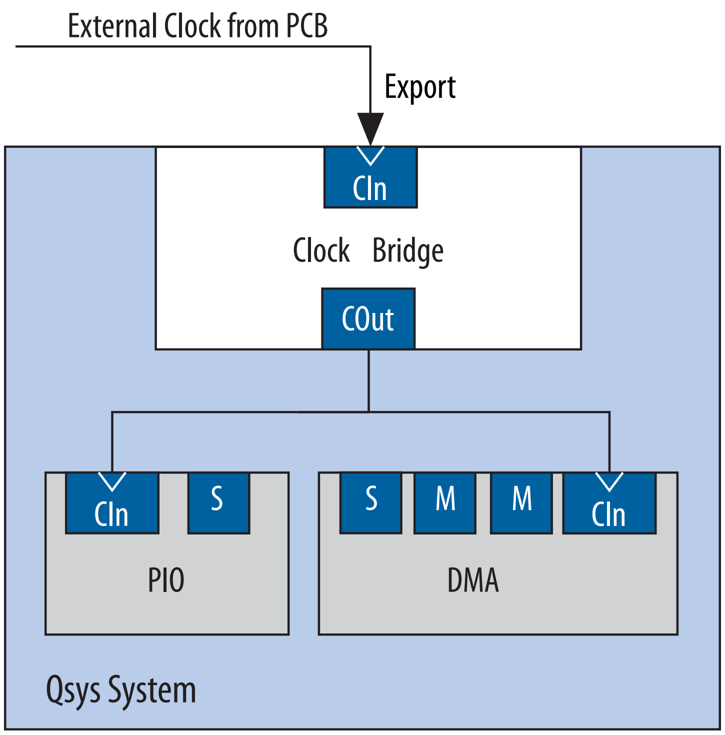

M1
M2
M3
M
M
M
Arbiter & Write Data Control Signal Multiplexing
S
Bridge
M
ChipSelect & Read Data Multiplexing
S
S
S
S1
S2
S3
M Master
S Slave

Clock Bridge

The Clock Bridge allows you to connect a clock source to multiple clock input interfaces. You can use the clock bridge to connect a clock source that is outside the Qsys system. You create the connection through an exported interface, and then connect to multiple clock input interfaces.

Clock outputs have the ability to fan-out without the use of a bridge. You require a bridge only when you want a clock from an exported source to connect internally to more than one source.

Figure 10-2: Clock Bridge



Avalon-MM Clock Crossing Bridge

The Avalon-MM Clock Crossing Bridge transfers Avalon-MM commands and responses between different clock domains. You can also use the Avalon-MM Clock Crossing Bridge between AXI masters and slaves of different clock domains.

The Avalon-MM Clock Crossing Bridge uses asynchronous FIFOs to implement clock crossing logic. The bridge parameters control the depth of the command and response FIFOs in both the master and slave clock domains. If the number of active reads exceeds the depth of the response FIFO, the Clock Crossing Bridge stops sending reads.

To maintain throughput for high-performance applications, increase the response FIFO depth from the default minimum depth, which is twice the maximum burst size.

Note: When you use the FIFO-based clock crossing a Qsys system, the DC FIFO is automatically inserted in the Qsys system. The reset inputs for the DC FIFO are connected to the reset sources for the connected master and slave components on either side of the DC FIFO. With this configuration, both the master side and slave side resets must be asserted at the same time to ensure that the DC

FIFO is reset properly. Alternatively, you can drive both resets from the same reset source to guarantee that the DC FIFO is reset properly.

Note: The clock crossing bridge includes appropriate SDC constraints for its internal asynchronous FIFOs. For these SDC constraints to work correctly, you should not set false paths on the pointer crossings in the FIFOs. You should also not split the bridge's clocks into separate clock groups when you declare SDC constraints; this has the same effect as setting false paths.

Related Information

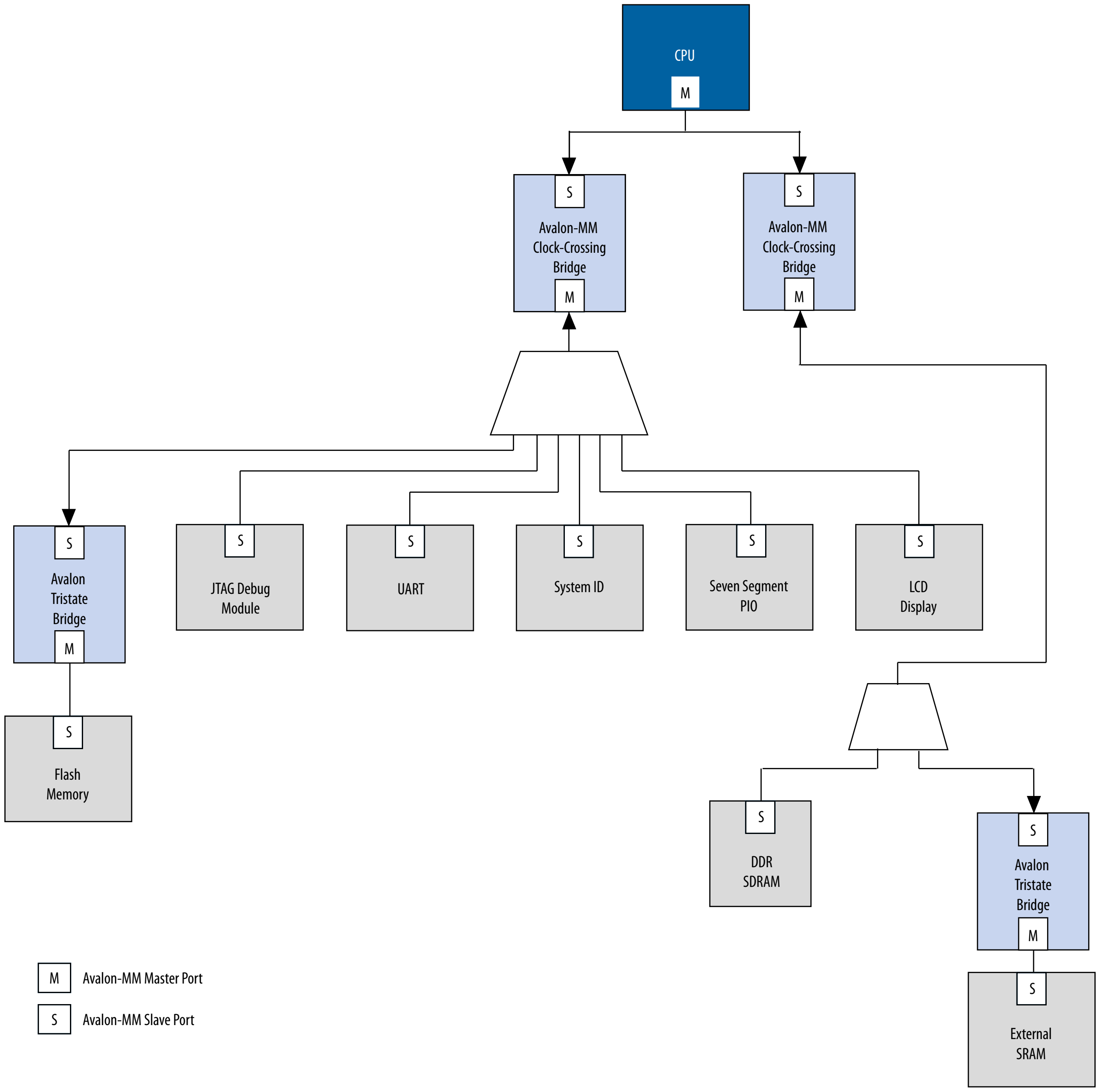
- **Creating a System with Qsys** on page 5-1

Avalon-MM Clock Crossing Bridge Example

In the example shown below, the Avalon-MM Clock Crossing bridges separate slave components into two groups. Low-performance slave components are placed behind a single bridge and are clocked at a low speed. High performance components are placed behind a second bridge and are clocked at a higher speed.

By inserting clock-crossing bridges, you simplify the Qsys interconnect and allow the Quartus II Fitter to optimize paths that require minimal propagation delay.

Figure 10-3: Avalon-MM Clock Crossing Bridge



Avalon-MM Clock Crossing Bridge Parameters

Table 10-1: Avalon-MM Clock Crossing Bridge Parameters

Parameters	Values	Description
Data width	8, 16, 32, 64, 128, 256,512, 1024 bits	Determines the data width of the interfaces on the bridge, and affects the size of both FIFOs. For the highest bandwidth, set **Data width** to be as wide as the widest master that connects to the bridge.
Symbol width	1, 2, 4, 8, 16, 32, 64 (bits)	Number of bits per symbol. For example, byte-oriented interfaces have 8-bit symbols.
Address width	1-32 bits	The address bits needed to address the downstream slaves.
Use automatically-determined address width	-	The minimum bridge address width that is required to address the downstream slaves.
Maximum burst size	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 bits	Determines the maximum length of bursts that the bridge supports.
Command FIFO depth	2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 2048, 4096, 8192, 16384 bits	Command (master-to-slave) FIFO depth.
Respond FIFO depth	2, 4, 8,16, 32, 64, 128, 256, 512, 1024 2048, 4096, 8192,16384 bits	Response (slave-to-master) FIFO depth.
Master clock domain synchronizer depth	2, 3, 4, 5 bits	The number of pipeline stages in the clock crossing logic in the issuing master to target slave direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a design by running a TimeQuest timing analysis.

Parameters	Values	Description
Slave clock domain synchronizer depth	2, 3, 4, 5 bits	The number of pipeline stages in the clock crossing logic in the target slave to master direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a design by running a TimeQuest timing analysis.

Avalon-MM Pipeline Bridge

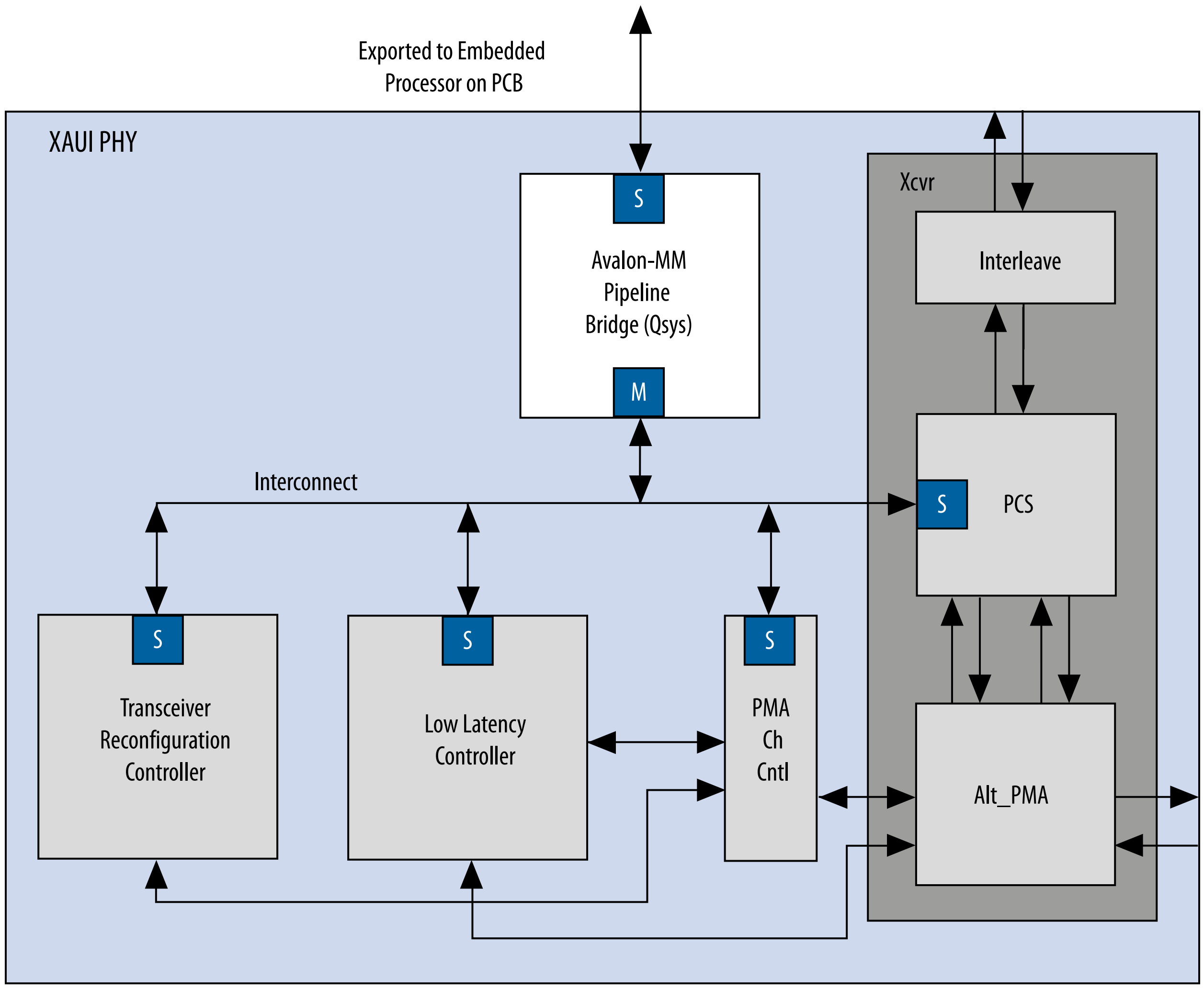
The Avalon-MM Pipeline Bridge inserts a register stage in the Avalon--MM command and response paths. The bridge accepts commands on its slave port and propagates the commands to its master port. The pipeline bridge provides separate parameters to turn on pipelining for command and response signals.

The **Maximum pending read transactions** parameter is the maximum number of pending reads that the Avalon-MM bridge can queue up. To determine the best value for this parameter, review this same option for the bridge's connected slaves and identify the highest value of the parameter, and then add the internal buffering requirements of the Avalon-MM bridge. In general, the value should be between 4 and 32. The limit for maximum queued transactions is 64.

You can use the Avalon-MM bridge to export a single Avalon-MM slave interface to use to control multiple Avalon-MM slave devices. The pipelining feature is optional. You can optionally turn off the pipelining feature of this bridge.

Figure 10-4: Avalon-MM Pipeline Bridge in a XAUI PHY Transceiver IP Core

In this example, the bridge transfers commands received on its slave interface to its master port.

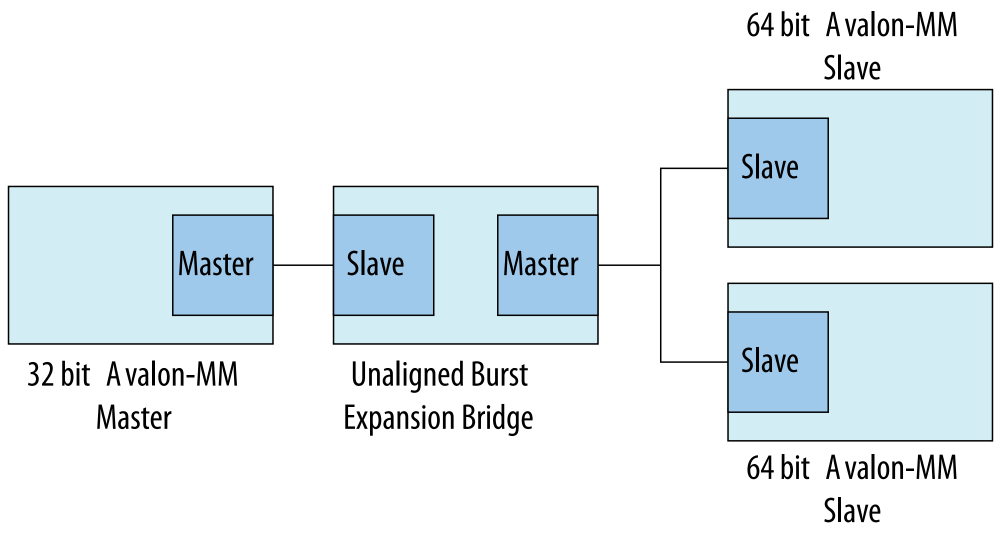

Exported to Embedded Processor on PCB
XAUI PHY
S
Avalon-MM Pipeline Bridge (Qsys)
M
Xcvr
Interleave
Interconnect
S
PCS
S
Transceiver Reconfiguration Controller
S
Low Latency Controller
S
PMA Ch Cntl
Alt_PMA

Because the slave interface is exported to the pins of the device, having a single slave port, rather than separate ports for each slave device, reduces the pin count of the FPGA.

Avalon-MM Unaligned Burst Expansion Bridge

The Avalon-MM Unaligned Burst Expansion Bridge aligns read burst transactions from masters connected to its slave interface, to the address space of slaves connected to its master interface. This alignment ensures that all read burst transactions are delivered to the slave as a single transaction.

Figure 10-5: Avalon-MM Unaligned Burst Expansion Bridge



You can use the Avalon Unaligned Burst Expansion Bridge to align read burst transactions from masters that have narrower data widths than the target slaves. Using the bridge for this purpose improves bandwidth utilization for the master-slave pair, and ensures that un-aligned bursts are processed as single transactions rather than multiple transactions.

Note: Do not use the Avalon-MM Unaligned Burst Expansion Bridge if any connected slave has read side effects from reading addresses that are exposed to any connected master's address map. This bridge can cause read side effects due to alignment modification to read burst transaction addresses.

Note: For Qsys 14.0, the Avalon-MM Unaligned Burst Expansion Bridge does not support VHDL simulation.

Related Information

- **Qsys Interconnect** on page 7-1

Using the Avalon-MM Unaligned Burst Expansion Bridge

When a master sends a read burst transaction to a slave, the Avalon-MM Unaligned Burst Expansion Bridge initially determines whether the start address of the read burst transaction is aligned to the slave's memory address space. If the base address is aligned, the bridge does not change the base address. If the base address is not aligned, the bridge aligns the base address to the nearest aligned address that is less than the requested base address.

The Avalon-MM Unaligned Burst Expansion Bridge then determines whether the final word requested by the master is the last word at the slave read burst address. If a single slave address contains multiple words, all of those words must be requested in order for a single read burst transaction to occur.

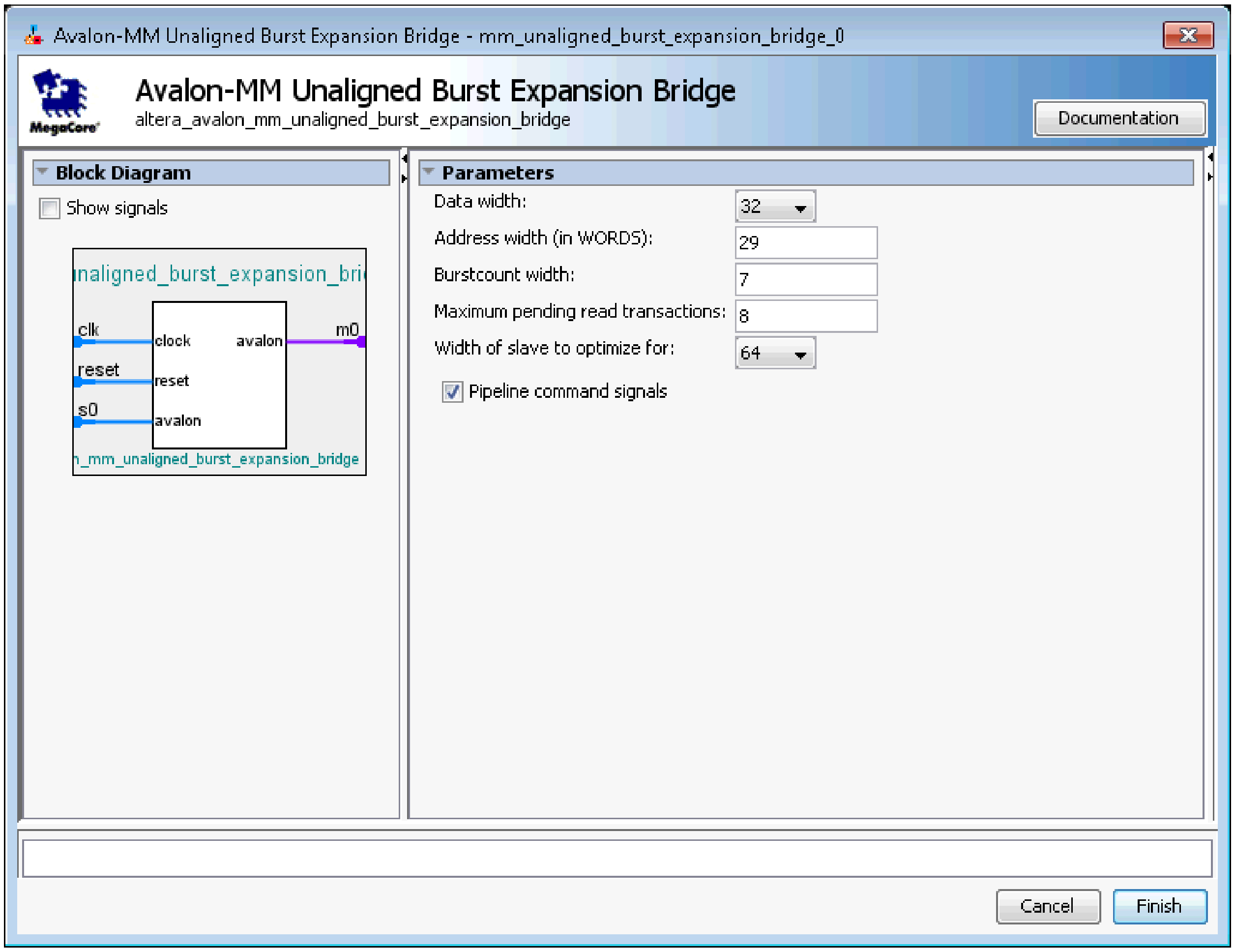
- If the final word requested by the master is the last word at the slave read burst address, the bridge does not modify the burst length of the read burst command to the slave.
- If the final word requested by the master is not the last word at the slave read burst address, the bridge increases the burst length of the read burst command to the slave. The final word requested by the modified read burst command is then the last word at the slave read burst address.

The bridge stores information about each aligned read burst command that it sends to slaves connected to a master interface. When a read response is received on the master interface, the bridge determines if the base address or burst length of the issued read burst command was altered.

If the bridge alters either the base address or the burst length of the issued read burst command, it receives response words that the master did not request. The bridge suppresses words that it receives from the aligned burst response that are not part of the original read burst command from the master.

Avalon-MM Unaligned Burst Expansion Bridge Parameters

Figure 10-6: Avalon-MM Unaligned Burst Expansion Bridge Parameter Editor


Avalon-MM Unaligned Burst Expansion Bridge - mm_unaligned_burst_expansion_bridge_0
Avalon-MM Unaligned Burst Expansion Bridge
altera_avalon_mm_unaligned_burst_expansion_bridge
Documentation
Block Diagram
Show signals
naligned_burst_expansion_bri
clk
clock
avalon
m0
reset
reset
s0
avalon
_mm_unaligned_burst_expansion_bridge
Parameters
Cancel
Finish

Data width: 32
Address width (in WORDS): 29
Burstcount width: 7
Maximum pending read transactions: 8
Width of slave to optimize for: 64
[✓] Pipeline command signals

Table 10-2: Avalon-MM Unaligned Burst Expansion Bridge Parameters

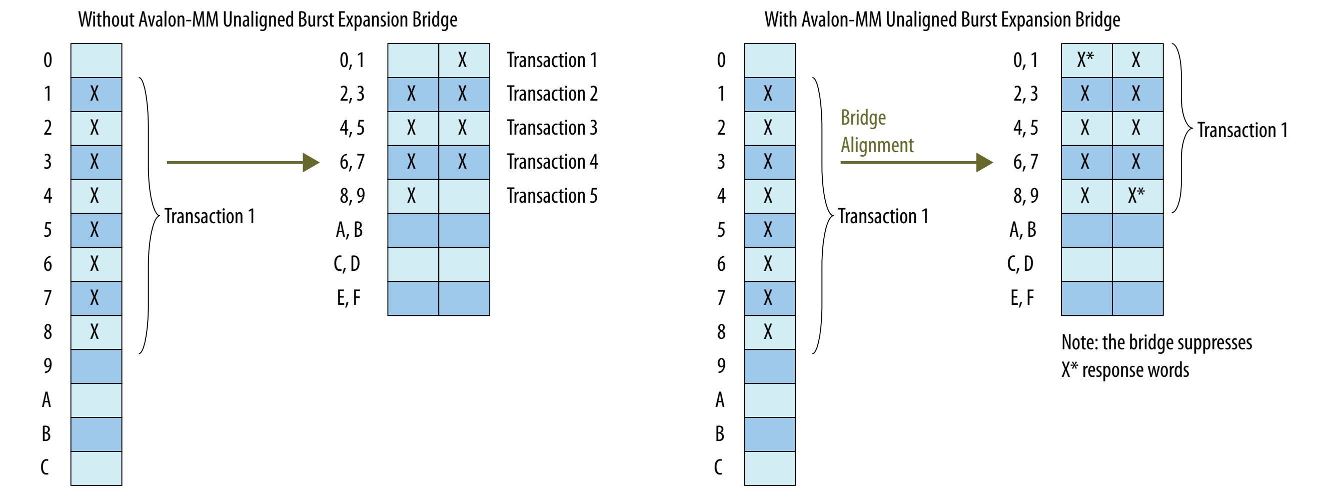
Parameter	Description
Data width	Data width of the master connected to the bridge.
Address width (in WORDS)	The address width of the master connected to the bridge.
Burstcount width	The burstcount signal width of the master connected to the bridge.

Parameter	Description
Maximum pending read transactions	The **Maximum pending read transactions** parameter is the maximum number of pending reads that the Avalon-MM bridge can queue up. To determine the best value for this parameter, review this same option for the bridge's connected slaves and identify the highest value of the parameter, and then add the internal buffering requirements of the Avalon-MM bridge. In general, the value should be between 4 and 32. The limit for maximum queued transactions is 64.
Width of slave to optimize for	The data width of the connected slave. Supported values are: 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 bits. **Note:** If you connect multiple slaves, all slaves must have the same data width.
Pipeline command signals	When turned on, the command path is pipelined, minimizing the bridge's critical path at the expense of increased logic usage and latency.

Avalon-MM Unaligned Burst Expansion Bridge Example

Figure 10-7: Unaligned Burst Expansion Bridge

The example below shows an unaligned read burst command from a master that the Avalon-MM Unaligned Burst Expansion Bridge converts to an aligned request for a connected slave, and the suppression of words due to the aligned read burst command. In this example, a 32-bit master requests an 8-beat burst of 32-bit words from a 64-bit slave with a start address that is not 64-bit aligned.


Without Avalon-MM Unaligned Burst Expansion Bridge
0
1
2
3
4
5
6
7
8
9
A
B
C
X
Transaction 1
0, 1
2, 3
4, 5
6, 7
8, 9
A, B
C, D
E, F
Transaction 1
Transaction 2
Transaction 3
Transaction 4
Transaction 5
With Avalon-MM Unaligned Burst Expansion Bridge
Bridge Alignment
X*
Note: the bridge suppresses X* response words

Because the target slave has a 64-bit data width, address 1 is unaligned in the slave's address space. As a result, several smaller burst transactions are needed to request the data associated with the master's read burst command.

With an Avalon-MM Unaligned Burst Expansion Bridge in place, the bridge issues a new read burst command to the target slave beginning at address 0 with burst length 10, which requests data up to the word stored at address 9.

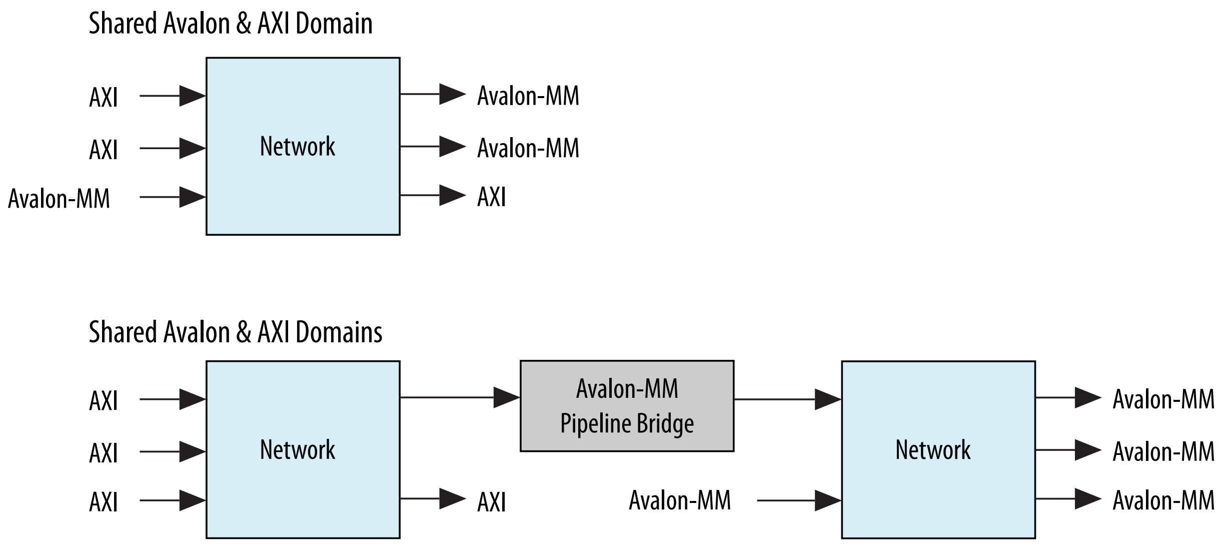
When the bridge receives the word corresponding to address 0, it suppresses it from the master, and then delivers the words corresponding to addresses 1 through 8 to the master. When the bridge receives the word corresponding to address 9, it suppresses that word from the master.

Bridges Between Avalon and AXI Interfaces

When designing a Qsys system, you can make connections between AXI and Avalon interfaces without the use of explicitly-instantiated bridges; the interconnect provides all necessary bridging logic. However, this does not prevent the use of explicit bridges to separate the AXI and Avalon domains.

Figure 10-8: Avalon-MM Pipeline Bridge Between Avalon-MM and AXI Domains

Using an explicit Avalon-MM bridge to separate the AXI and Avalon domains reduces the amount of bridging logic in the interconnect at the expense of concurrency.



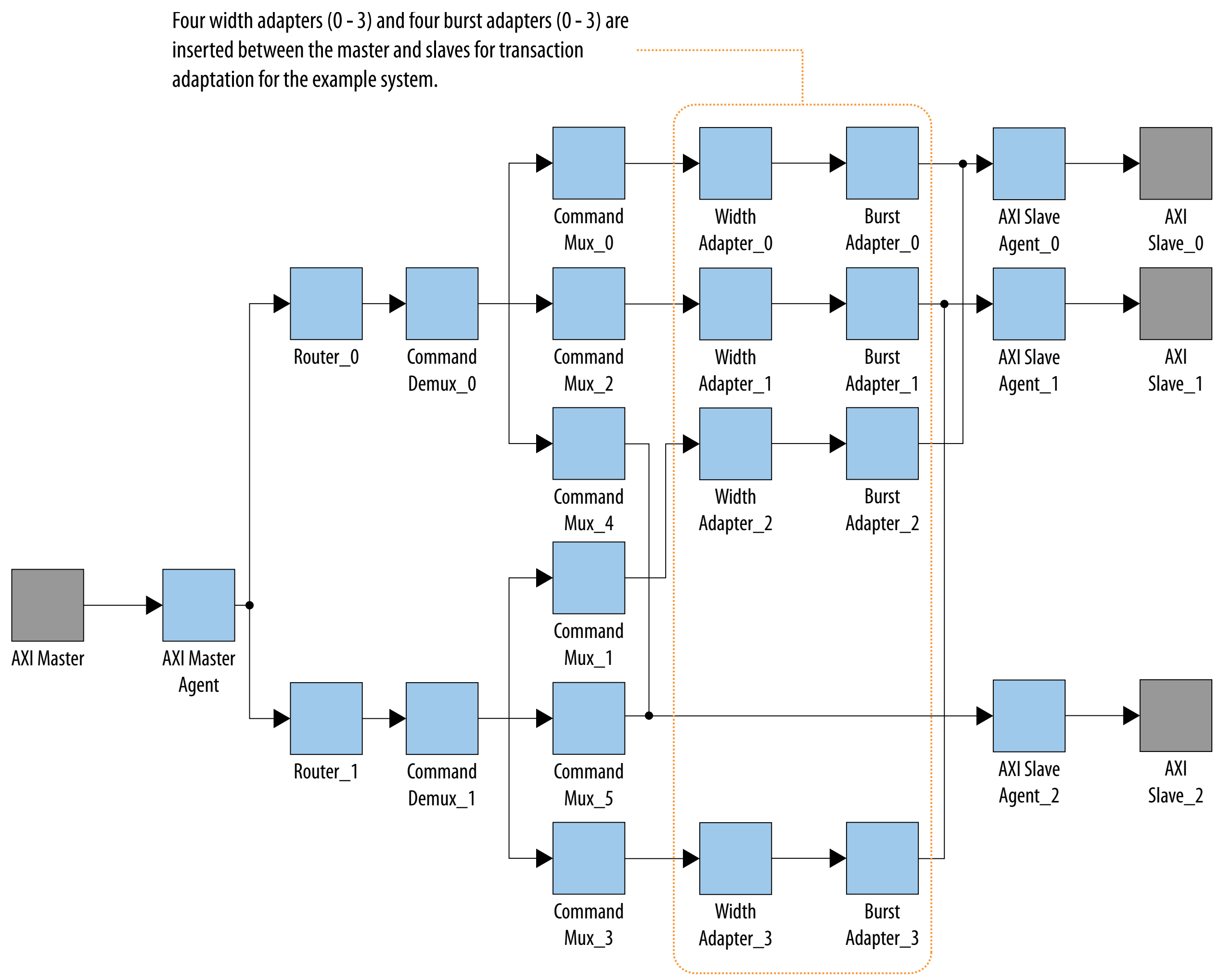
AXI Bridge

With an AXI bridge, you can influence the placement of resource-intensive components, such as the width and burst adapters. Depending on its use, an AXI bridge may reduce throughput and concurrency, in return for higher f_{Max} and less logic.

You can use an AXI bridge to group different parts of your Qsys system. Then, other parts of the system connect to the bridge interface instead of to multiple separate master or slave interfaces. You can also use an AXI bridge to export AXI interfaces from Qsys systems.

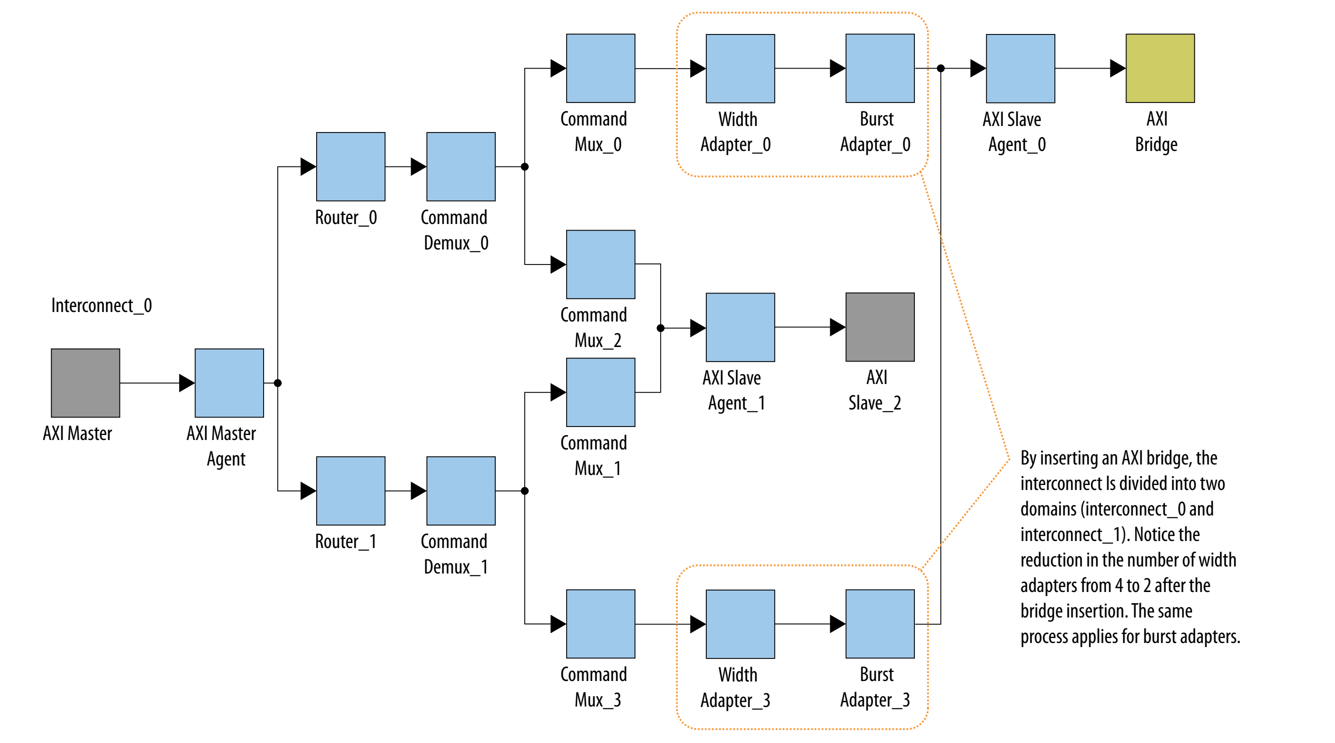
The example below shows a system with a single AXI master and three AXI slaves. It also has various interconnect components, such as routers, demuxes, and muxes. Two of the slaves have a narrower data width than the master; 16-bit slaves versus a 32-bit master. In this system, Qsys interconnect creates four width adapters and four burst adapters to access the two slaves. In this case, you could improve resource usage by adding an AXI bridge. This would result in Qsys having to add only two width adapters and two burst adapters, one pair for the read channels, and another pair for the write channel.

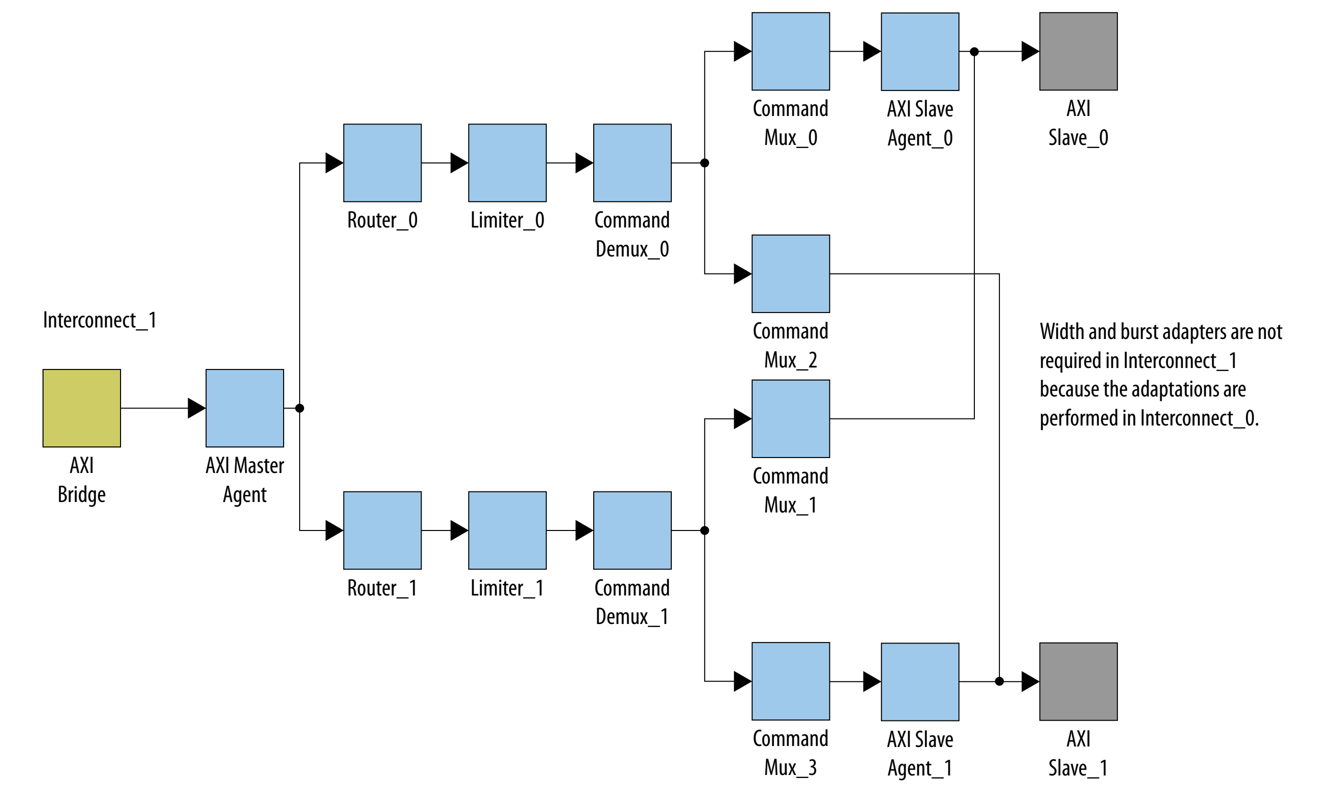
Figure 10-9: AXI Example Without a Bridge: Adding a Bridge Can Reduce the Number of Adapters



The example below shows the same system with an AXI bridge component, and the decrease in the number of width and burst adapters. Qsys creates only two width adapters, and two burst adapters, as compared to the four width adapters and four burst adapters in the previous example. The system includes more components, but the overall system performance improves because there are fewer resource-intensive width and burst adapters.

Figure 10-10: Width and Burst Adapters Added to a System With a Bridge





AXI Bridge Signal Types

Based on parameter selections that you make for the AXI Bridge component, Qsys instantiates either the AXI3 or AXI4 master and slave interfaces into the component.

Note: In AXI3, `aw/aruser` accommodates sideband signal usage by hard processor systems (HPS).

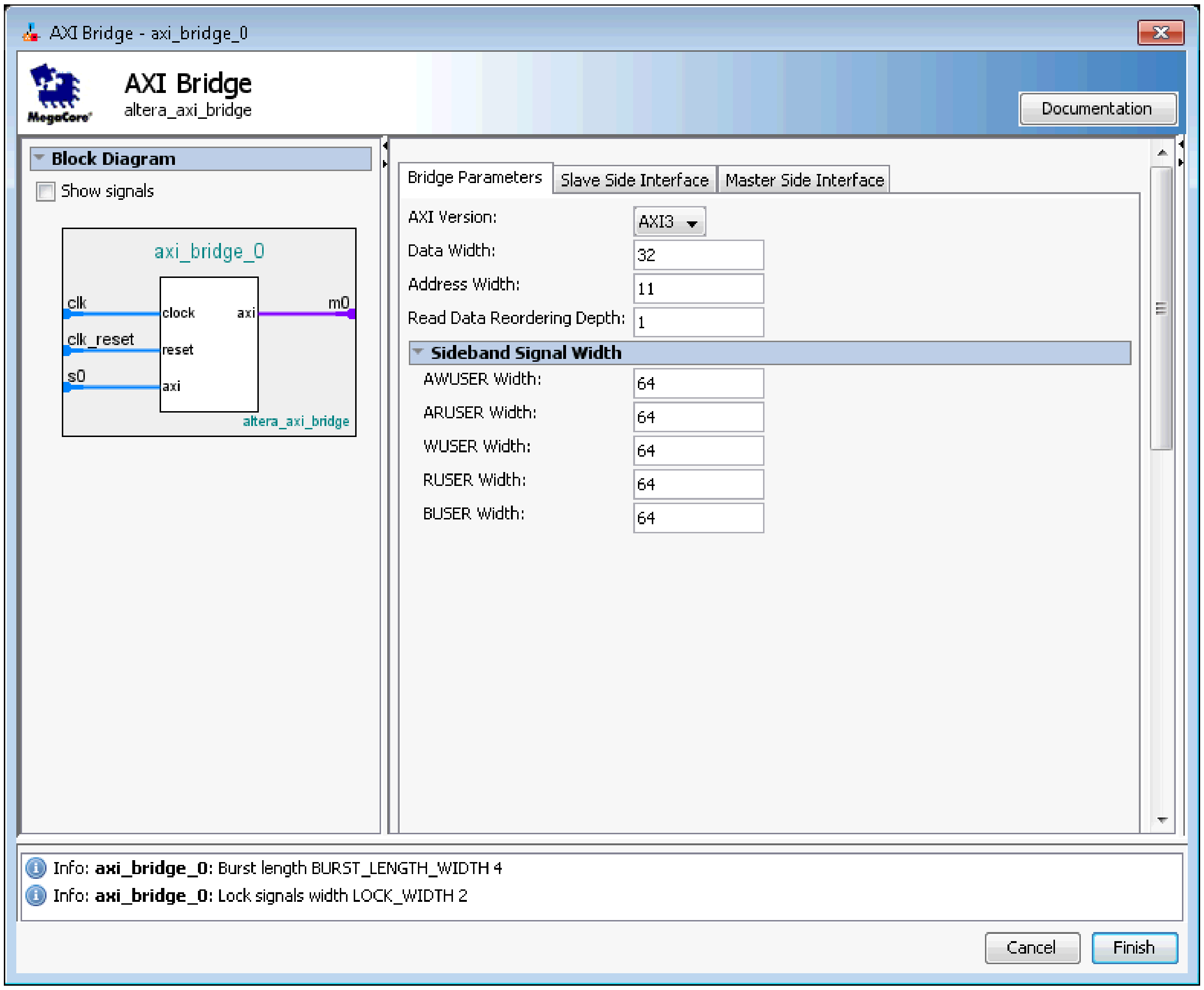
Table 10-3: Sets of Signals for the AXI Bridge Based on the Protocol

Signal Name	AXI3	AXI4
`awid / arid`	yes	yes
`awaddr /araddr`	yes	yes
`awlen / arlen`	yes (4-bit)	yes (8-bit)
`awsize/ arsize`	yes	yes
`awburst /arburst`	yes	yes
`awlock /arlock`	yes	yes (1-bit optional)
`awcache / arcache`	yes (2-bit)	yes (optional)
`awprot / arprot`	yes	yes
`awuser /aruser`	yes	yes
`awvalid / arvalid`	yes	yes
`awready /arready`	yes	yes
`awqos /arqos`	no	yes
`awregion /arregion`	no	yes
`wid`	yes	no (optional)
`wdata / rdata`	yes	yes
`wstrb`	yes	yes
`wlast /rvalid`	yes	yes
`wvalid /rlast`	yes	yes
`wready /rready`	yes	yes
`wuser / ruser`	no	yes
`bid / rid`	yes	yes
`bresp / rresp`	yes	yes (optional)
`bvalid`	yes	yes
`bready`	yes	yes

AXI Bridge Parameters

In the parameter editor, you can customize the parameters for the AXI bridge according to the requirements of your design.

Figure 10-11: AXI Bridge Parameter Editor



AXI Version: AXI3
Data Width: 32
Address Width: 11
Read Data Reordering Depth: 1

Sideband Signal Width
AWUSER Width: 64
ARUSER Width: 64
WUSER Width: 64
RUSER Width: 64
BUSER Width: 64

Info: **axi_bridge_0:** Burst length BURST_LENGTH_WIDTH 4
Info: **axi_bridge_0:** Lock signals width LOCK_WIDTH 2

Table 10-4: AXI Bridge Parameters

Parameter	Type	Range	Description
AXI Version	string	AXI3/ AXI4	Specifies the AXI version and signals that Qsys generates for the slave and master interfaces of the bridge.
Data Width	int	8:1024	Controls the width of the data for the master and slave interfaces.
Address Width	int	1-64 bits	Controls the width of the address for the master and slave interfaces.

Parameter	Type	Range	Description
Read Data Reordering Depth	int	1-16	Controls the multithreading feature and out-of-order responses. If a master issues different thread IDs to different slaves, in order for a slave to view the different thread IDs, you must set the **Read Data Reordering Depth** to 1.
AWUSER Width	int	1-64 bits	Controls the width of the write address channel sideband signals of the master and slave interfaces.
ARUSER Width	int	1-64 bits	Controls the width of the read address channel sideband signals of the master and slave interfaces.
WUSER Width	int	1-64 bits	Controls the width of the write data channel sideband signals of the master and slave interfaces.
RUSER Width	int	1-16 bits	Controls the width of the read data channel sideband signals of the master and slave interfaces.
BUSER Width	int	1-16 bits	Controls the width of the write response channel sideband signals of the master and slave interfaces.

AXI Bridge Slave and Master Interface Parameters

Table 10-5: AXI Bridge Slave and Master Interface Parameters

Parameter	Description
ID Width	Controls the width of the thread ID of the master and slave interfaces.
Write/Read/Combined Acceptance Capability	Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the slave interface accepts.

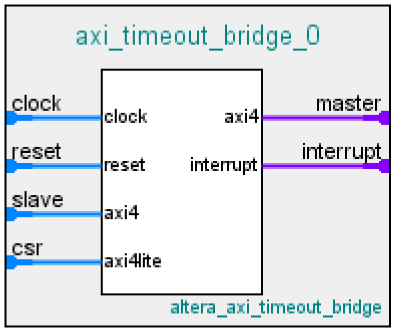
Parameter	Description
Write/Read/Combined Issuing Capability	Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the master interface issues. Issuing capability must follow acceptance capability to avoid unnecessary creation of FIFOs in the bridge.

Note: Maximum acceptance/issuing capability is a model-only parameter and does not influence the bridge HDL. The bridge does not backpressure when this limit is reached. Downstream components and/or the interconnect must apply backpreasure.

AXI Timeout Bridge

You can place an AXI Timeout Bridge between a single master and a single slave if you know that the slave may timeout and cause your system to hang. If a slave does not accept a command or respond to a command it accepted, its master can wait indefinitely. The AXI Timeout Bridge allows your system to recover when it hangs, and can also facilitate debugging.

Figure 10-12: AXI Timeout Bridge

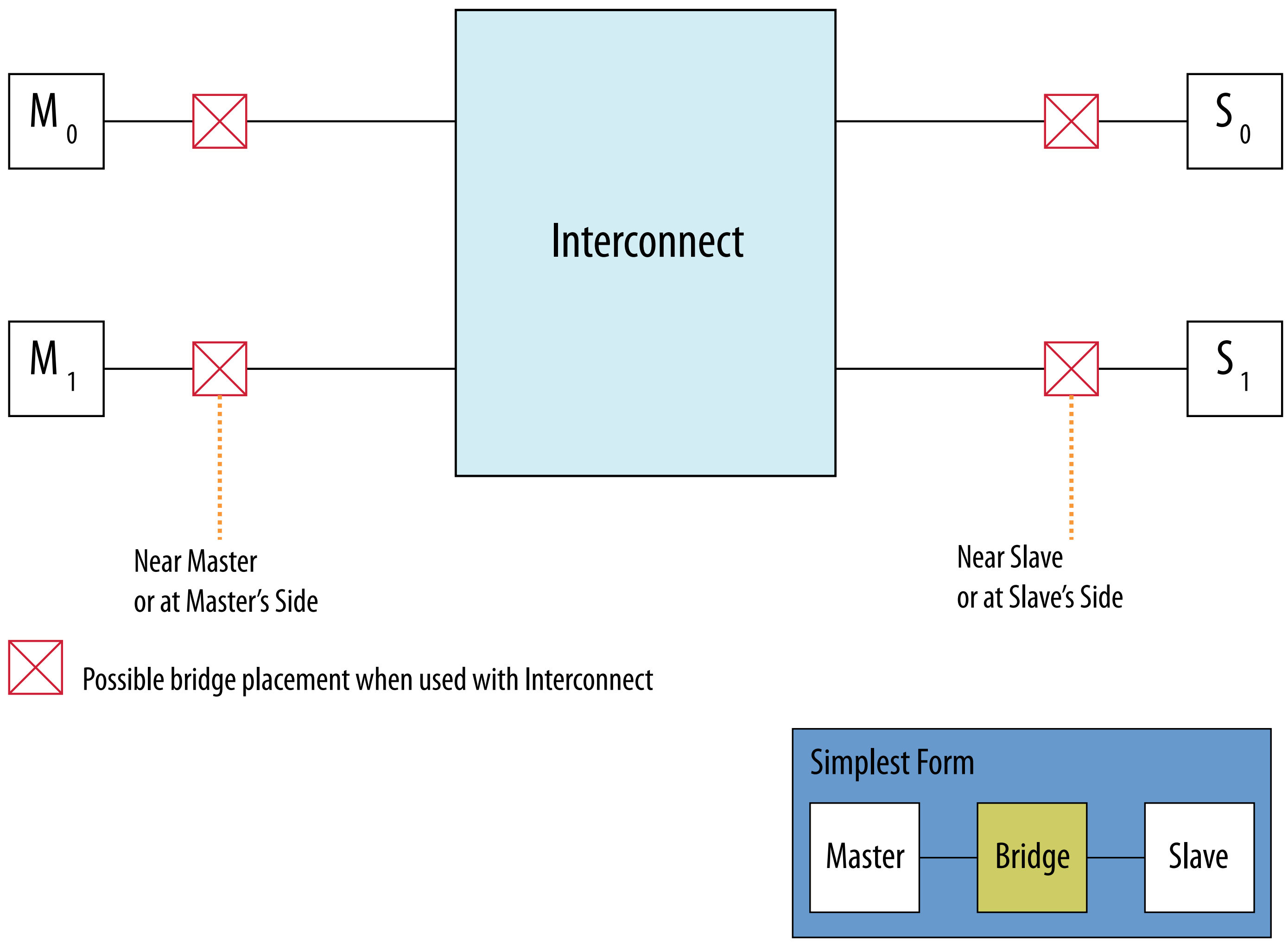


For a domain with multiple masters and slaves, placement of an AXI Timeout Bridge in your design may be beneficial in the following scenarios:

- To recover from a hang, place the bridge near the slave. If the master attempts to communicate with a slave that hangs, the AXI Timeout Bridge frees the master by generating error responses. The master is then able to communicate with another slave.
- When debugging your system, place the AXI Timeout Bridge near the master. This placement enables you to identify the origin of the burst and to obtain the full address from the master. Additionally, placing an AXI Timeout Bridge near the master enables you to identify the target slave for the burst.

  Note: If you put the bridge at the slave's side and you have multiple slaves connected to the same master, you do not get the full address.

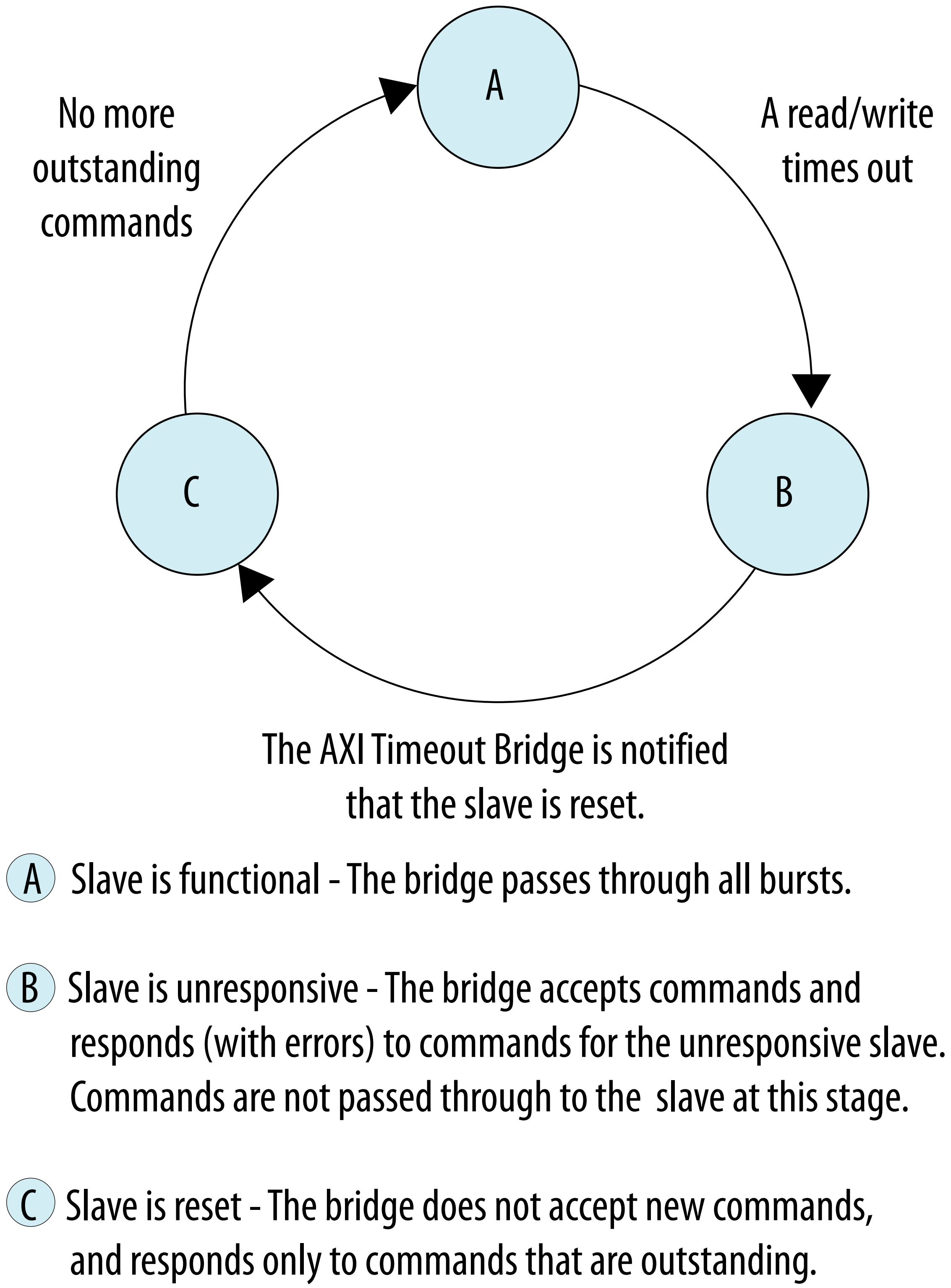
Figure 10-13: AXI Timeout Bridge Placement



AXI Timeout Bridge Stages

A timeout occurs when the internal timer in the bridge exceeds the specified number of cycles within which a burst must complete from start to end.

Figure 10-14: AXI Bridge Timeout Bridge Stages


A
A read/write times out
B
The AXI Timeout Bridge is notified that the slave is reset.
C
No more outstanding commands

A Slave is functional - The bridge passes through all bursts.

B Slave is unresponsive - The bridge accepts commands and responds (with errors) to commands for the unresponsive slave. Commands are not passed through to the slave at this stage.

C Slave is reset - The bridge does not accept new commands, and responds only to commands that are outstanding.

When a timeout occurs, the AXI Timeout Bridge asserts an interrupt and reports the burst that caused the timeout to the CSR. The bridge then generates error responses back to the master on behalf of the unresponsive slave. This stage frees the master and certifies the unresponsive slave as dysfunctional. The AXI Timeout Bridge then accepts subsequent write addresses, write data and read addresses to the dysfunctional slave. The bridge does not accept outstanding write responses and read data from the dysfunctional slave are not passed through to the master. The `awvalid`, `wvalid`, `bready`, `arvalid`, and `rready` ports are "held low" at the master interface of the bridge.

Note: After a timeout, `awvalid`, `wvalid` and `arvalid` may be dropped before they are accepted by `awready` at the master interface. While the behavior violates the AXI specification, it occurs only on an interface connected to the slave which has been certified dysfunctional by the AXI Timeout Bridge.

Write channel refers to the AXI write address, data and response channels. Similarly, read channel refers to the AXI read address and data channels. AXI write and read channels are independent of each other. However, when a timeout occurs on either channel, the bridge generates error responses on both channels.

Table 10-6: Burst Start and End Definitions for the AXI Timeout Bridge

Channel	Start	End
Write	When an address is issued. First cycle of `awvalid`, even if data of the same burst is issued before the address (first cycle of `wvalid`).	When the response is issued. First cycle of `bvalid`.
Read	When an address is issued. First cycle of `arvalid`.	When the last data is issued. First cycle of `rvalid` and `rlast`.

The AXI Timeout Bridge has four required interfaces: Master, Slave, Configuration and Status Register (CSR) (AXI4-Lite), and Interrupt. Qsys allows the AXI Timeout bridge to connect to any AXI3, AXI4, or Avalon master or slave interface. Avalon masters must to utilize the bridge's interrupt output to detect a timeout.

The bridge slave interface accepts write addresses, write data, and read addresses, and then generates the `SLVERR` response at the write response and read data channels. You should not expect to use `buser`, `rdata` and `ruser` at this stage of processing.

To resume normal operation, the dysfunctional slave must be reset and the bridge notified of the change in status via the CSR. Once the CSR notifies the bridge that the slave is ready, the bridge does not accept new commands until all outstanding bursts are responded to with an error response.

The CSR has a 4-bit address width, and a 32-bit data width. The CSR reports status and address information when the bridge asserts an interrupt.

Table 10-7: CSR Interrupt Status Information for the AXI Timeout Bridge

Address	Attribute	Name	Description
0x0	write-only	Slave is reset	Write a 1 to notify the AXI Timeout Bridge that the slave is reset and ready. Clears the interrupt.
0x4	read-only	Timed out operation	The operation of the burst that caused the timeout. 1 for a write; 0 for a read.
0x8 through 0xf	read-only	Timed out address	The address of the burst that caused the timeout. For an address width greater than 32-bits, CSR reads addresses 0x8 and 0xc to obtain the complete address.

AXI Timeout Bridge Parameters

Table 10-8: AXI Timeout Bridge Parameters

Parameter	Description
ID width	The width of `awid`, `bid`, `arid`, or `rid`.
Address width	The width of `awaddr` or `araddr`.
Data width	The width of `wdata` or `rdata`.
User width	The width of `awuser`, `wuser`, `buser`, `aruser`, or `ruser`.
Maximum number of outstanding writes	The expected maximum number of outstanding writes.
Maximum number of outstanding reads	The expected maximum number of outstanding reads.
Maximum number of cycles	The number of cycles within which a burst must complete.

Address Span Extender

The Address Span Extender creates a windowed bridge and allows memory-mapped master interfaces to access a larger or smaller address map than the width of their address signals allow. With an address span extender, a restricted master can access a broader address range. The address span extender splits the addressable space into multiple separate windows so that the master can access the appropriate part of the memory through the window.

The address span extender does not limit master and slave widths to a 32-bit and 64-bit configuration. You can use the address span extender for other width configurations. The address span extender supports 1-64 bit address windows.

If a processor can address only 2GB of an address span, and your system contains 4GB of memory, the address span extender can provide two 2GB windows in the 4GB memory address space. This issue sometimes occurs with Altera SoC devices. For example, an HPS subsystem in an SoC device can address only 1GB of an address span within the FPGA using the HPS-to-FPGA bridge. The address span extender enables the SoC device to address all of the address space in the FPGA using multiple 1GB windows.

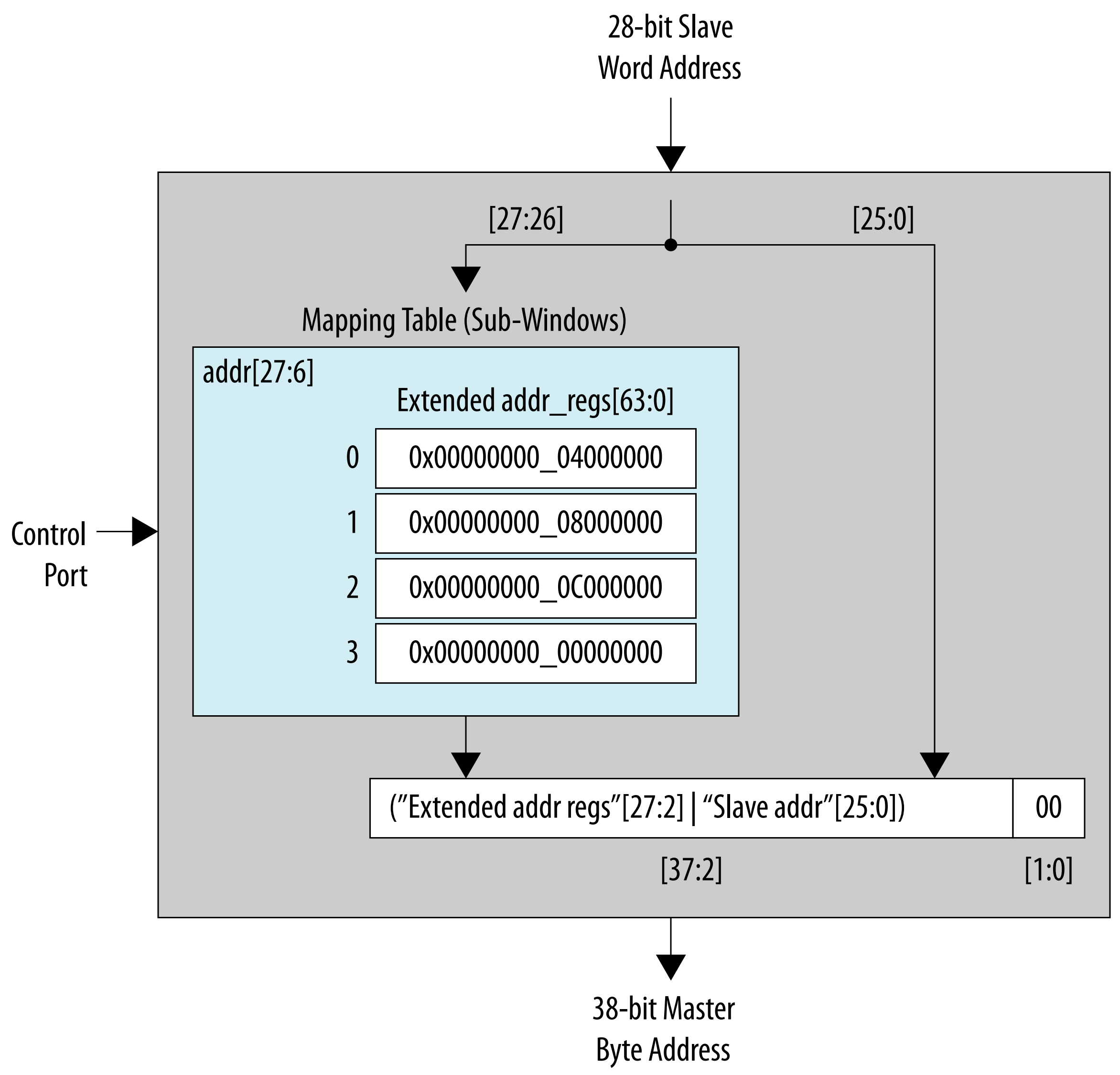
CTRL Register Layout

The control registers consist of a 64-bit register for each window. You write the base address that you want for each window to its corresponding control register. For example, if `CTRL_BASE` is the base address of the address span extender's control register, and there are two windows (0 and 1), then window 0’s control register starts at `CTRL_BASE`, and window 1’s control register starts at `CTRL_BASE` + 8 (using byte addresses).

Calculating the Address Span Extender Slave Address

The diagram below describes how Qsys calculates the slave address. In this example the address, span extender is configured with a 28-bit address space for slaves. The lower 26 bits (bits 0 to 25 or `[25:0]`) is the offset into a particular window and originate from the address span extender's data port. The upper 2 bits `[27:26]` originate from the control registers.

Figure 10-15: Address Span Extender



Using the Address Span Extender

When you implement the address span extender in Qsys, you must know the amount of address space the master uses (the size of the window), the total size of the addressable space (the number of windows), and how much address space (the size of the window) you want a particular slave to occupy in a master's address map.

This component supports 1 to 64 address windows. Qsys requires an assigned number of registers to hold the upper address bits for each window. In the parameter editor, you must select the number of bits in the expanded address map you want to access (**Expanded Master Byte Address Width**), the number of bits you want the master to see (**Slave Word Address Width**), and the number of sub-windows.

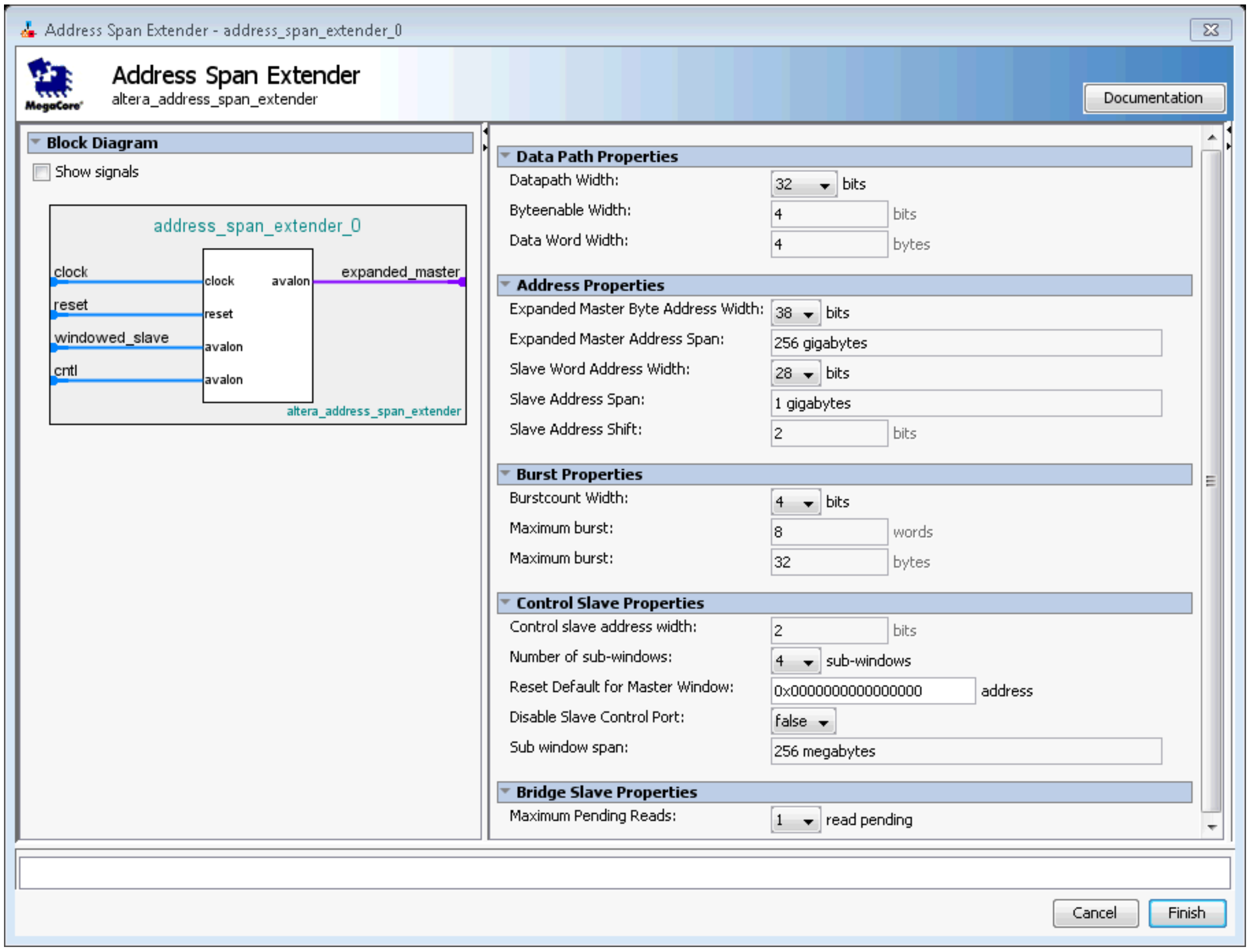
Each sub-window has a 64-bit register set that defines the sub window's upper address, and use only the bits greater than the slave byte address.

- **window 0**—expanded address `[63:0]`
- **window 1**—expanded address `[63:0]`

Qsys uses the upper bits of the slave address to pick which window to use. For example, if you specify 4 windows, then Qsys uses the top 2 bits of the slave address to specify window `[0,1,2,3]`. Therefore having more windows does require the windows to be smaller, for example having 4 windows requires the windows themselves to be 1/4 the size of the slave address space. The total windowed address space is still equal to the original slave address space, but the windows allow access to memory regions in a larger overall address space.

In the parameter editor for the address span extender, you can click **Documentation** to obtain more information about the component.

Figure 10-16: Address Span Extender Parameter Editor



Data Path Properties
- Datapath Width: 32 bits
- Byteenable Width: 4 bits
- Data Word Width: 4 bytes

Address Properties
- Expanded Master Byte Address Width: 38 bits
- Expanded Master Address Span: 256 gigabytes
- Slave Word Address Width: 28 bits
- Slave Address Span: 1 gigabytes
- Slave Address Shift: 2 bits

Burst Properties
- Burstcount Width: 4 bits
- Maximum burst: 8 words
- Maximum burst: 32 bytes

Control Slave Properties
- Control slave address width: 2 bits
- Number of sub-windows: 4 sub-windows
- Reset Default for Master Window: 0x0000000000000000 address
- Disable Slave Control Port: false
- Sub window span: 256 megabytes

Bridge Slave Properties
- Maximum Pending Reads: 1 read pending

Alternate Options for the Address Span Extender

You can set parameters for the address span extender with an initial fixed address value. Enter an address for the **Reset Default for Master Window** option, and select **True** for the **Disable Slave Control Port** option. This allows the address span extender to function as a fixed, non-programmable component.

Each sub-window is equal in size and stacks sequentially in the windowed slave interface's address space. To control the fixed address bits of a particular sub-window, you can write to the sub-window's register in the register control slave interface. Qsys structures the logic so that Qsys can optimize and remove bits that are not needed.

If **Burstcount Width** is greater than 1, Qsys processes the read burst in a single cycle, and assumes all byteenables are asserted on every cycle.

NIOS II Support

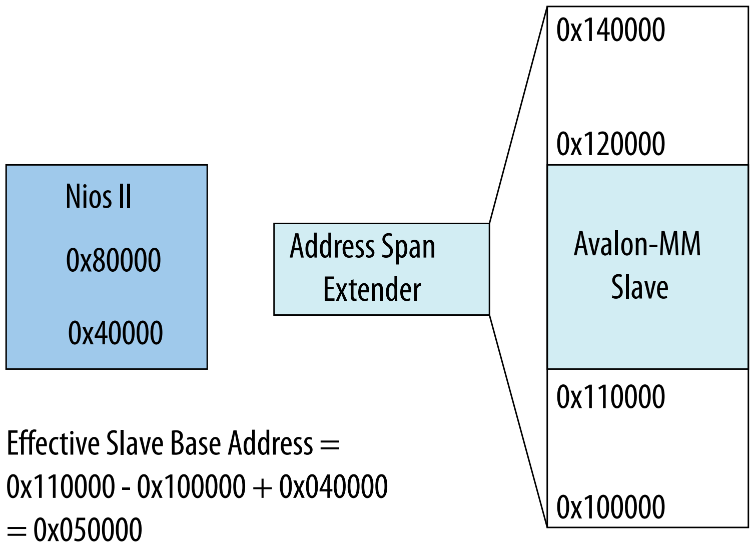
If the address span extender window is fixed, for example, the **Disable Slave Control Port** option is turned on, then the address span extender performs as a bridge. Components on the slave side of the address span extender that are within the window are visible to the NIOS II processor. Components partially within a window appear to NIOS II as if they have a reduced span. For example, a memory partially within a window appears as having a smaller size.

You can also use the address span extender to provide a window for the Nios II processor so that the HPS memory map is visible to NIOS II. In this way it is possible for the Nios II to communicate with HPS peripherals.

In the example below, a NIOS II processor has an address span extender from address `0x40000` to `0x80000`. There is a window within the address span extender starting at `0x100000`. Within the address span extender's address space there is a slave at base address `0x1100000`. The slave appears to NIOS II as being at address:

```
0x110000 - 0x100000 + 0x40000 = 0x050000
```

Figure 10-17: NIOS II Support and the Address Span Extender



If the address span extender window is dynamic. For example, when the **Disable Slave Control Port** option is turned off, the NIOS II processor is unable to see components on the slave side of the address span extender.

AXI Default Slave

An AXI Default Slave provides a predictable error response service for master interfaces that send transactions that attempt to access an undefined memory region. This service guarantees an error response, should a master access a memory region that is not decoded to an instantiated slave. The error response service also helps to avoid unpredictable behavior in your system.

The default slave is an AXI3 component and displays in the IP Catalog as either **AXI Default Slave** or **Error Response Slave**.

AXI protocol requires that if the interconnect cannot successfully decode slave access, it must return the `DECERR` error response. Therefore, the default slave is required in AXI systems where the address space is not fully decoded to slave interfaces.

The default slave behaves like any other component in the system and is bound by translation and adaptation interconnect logic. An increase in resource usage may occur when a default slave connects to masters of different data widths, including Avalon or AXI-Lite masters.

You can connect clock, reset, and IRQ signals to a default slave, as well as AXI3 and AXI4 master interfaces without also instantiating a bridge. When you connect a default slave to a master, the default slave accepts cycles sent from the master, and returns the `DECERR` error response. On the AXI interface, the default slave supports only a read and write acceptance of 1, and does not support write data interleaving. The read and write channels are independent, and responses are returned when simultaneously targeted by a read and write cycle.

There is an optional interface on the default slave that supports CSR accesses for debug. CSR registers log the required information when returning an error response. When turned on, this channel acts as an Avalon interface with read and write channels with a fixed latency of 1.

To enable a slave interface as a default slave for a master interface in your system, you must connect the slave to the master in your Qsys system. You specify a default slave for a master it by turning on the **Default Slave** column option in the **System Contents** tab. A system can contain more than one default slave. Altera recommends instantiating a separate default slave for each AXI master in your system.

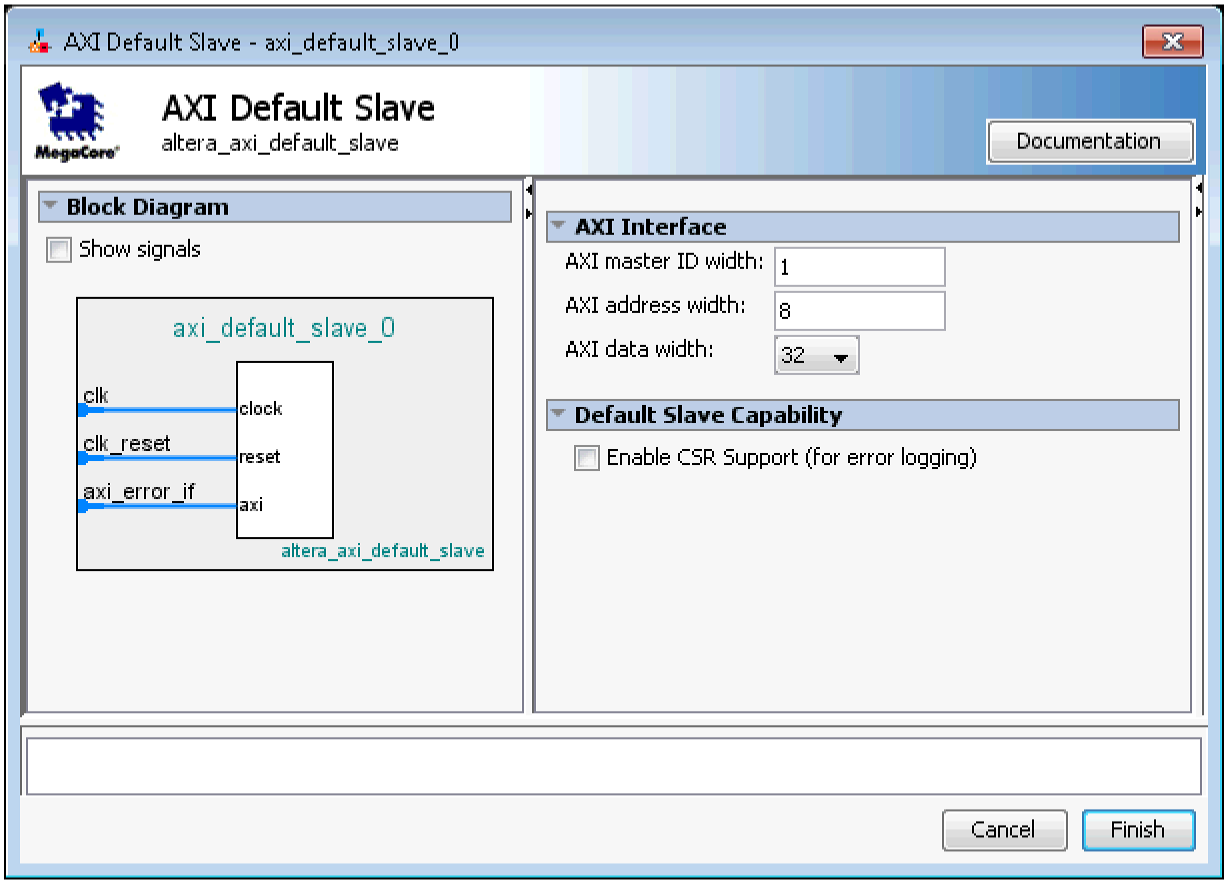
For information about creating secure systems and accessing undefined memory regions, refer to *Creating a System with Qsys* in volume 1 of the *Quartus II Handbook*.

Related Information

- **Creating a System with Qsys** on page 5-1

AXI Default Slave Parameters

Figure 10-18: AXI Default Slave Parameter Editor


AXI Default Slave - axi_default_slave_0

AXI Default Slave
altera_axi_default_slave

Block Diagram
Show signals: Off

AXI Interface
AXI master ID width: 1
AXI address width: 8
AXI data width: 32

Default Slave Capability
Enable CSR Support (for error logging): Off

Table 10-9: AXI Default Slave Parameters

Parameter	Value	Description
AXI master ID width	1-8 bits	Determines the master ID width for error logging.
AXI address width	8-64 bits	Determines the address width for error logging. This value also affects the overall address width of the system, and should not exceed the maximum address width required in the system.
AXI data width	32, 64, or128 bits	Determines the data width for error logging.
Enable CSR Support (for error logging)	On or Off	When turned on, instantiates an Avalon CSR interface for error logging.

Parameter	Value	Description
CSR Error Log Depth	1-16 bits	Depth of the transaction log, for example, the number of transactions the CSR logs for cycles with errors.
Register Avalon CSR inputs	On or Off	When turned on, controls debug access to the CSR interface.

CSR Registers

When an access violation occurs, and the CSR port is enabled, the AXI Default Slave generates an interrupt and transfers the transaction information into the error log FIFO.

The error log count continues until the n^{th} log, where *n* is the log depth. When Qsys responds to the interrupt bit, it reads the register until the interrupt bit is no longer valid. The interrupt bit is valid as long as there is a valid bit in FIFO. A cleared interrupt bit is not affected by the FIFO status. When Qsys finishes reading the register, the access violation service is ready to receive new access violation requests. If an access violation occurs when FIFO is full, then an overflow bit is set, indicating more than *n* access violations have occurred, and some are not logged.

Qsys exits the access violation service after either the interrupt bit is no longer set, or when it determines that the access violation service has continued for too long.

CSR Interrupt Status Registers

Table 10-10: CSR Interrupt Status Registers

For CSR register maps: `Address = Memory Address Base + Offset`.

Offset	Bit	Attribute	Default	Descripton
`0x00`	`31:4`	R0	0	Reserved.
	3	RW1C	0	**Read Access Violation Interrupt Overflow register** Asserted when a read access causes the Interconnect to return a `DECERR` response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.
	2	RW1C	0	**Write Access Violation Interrupt Overflow register** Asserted when a write access causes the Interconnect to return a `DECERR` response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.

Offset	Bit	Attribute	Default	Descripton
	1	RW1C	0	**Read Access Violation Interrupt register** Asserted when a read access causes the Interconnect to return a `DECERR` response. Cleared by setting the bit to 1. **Note:** Access violation are logged until the bit is cleared.
	0	RW1C	0	**Write Access Violation Interrupt register** Asserted when a write access causes the Interconnect to return a `DECERR` response. Cleared by setting the bit to 1. **Note:** Access violation are logged until the bit is cleared.

CSR Read Access Violation Log

The CSR read access violation log settings are valid only when an associated read interrupt register is set. This set of registers should be read until the valid bit is cleared.

Table 10-11: CSR Read Access Violation Log

Offset	Bit	Attribute	Default	Description
`0x100`	31:13	R0	0	Reserved.
	12:11	R0	0	Indicates the burst type of the initiating cycle that causes the access violation.
	10:7	R0	0	Indicates the burst length of the initiating cycle that causes the access violation.
	6:4	R0	0	Indicates the burst size of the initiating cycle that causes the access violation.
	3:1	R0	0	Indicates the `PROT` of the initiating cycle that causes the access violation.
	0	R0	0	Read access violation log for the transaction is valid only when this bit is set. This bit is cleared when the interrupt register is cleared.
`0x104`	31:0	R0	0	Master ID for the cycle that causes the access violation.
`0x108`	31:0	R0	0	Read cycle target address for the cycle that causes the access violation (lower 32-bit).

Offset	Bit	Attribute	Default	Description
`0x10C`	31:0	RO	0	Read cycle target address for the cycle that causes the access violation (upper 32-bit). Valid only if widest address in system is larger than 32 bits. **Note:** When this register is read, the current read access violation log is recovered from FIFO.

CSR Write Access Violation Log

The CSR write access violation log settings are valid only when an associated read interrupt register is set. This set of registers should be read until the valid bit is cleared.

Table 10-12: CSR Write Access Violation Log

Offset	Bit	Attribute	Default	Description
`0x190`	`31:13`	RO	0	Reserved.
	12:11	RO	0	Indicates the burst type of the initiating cycle that causes the access violation.
	10:7	RO	0	Indicates the burst length of the initiating cycle that causes the access violation.
	6:4	RO	0	Indicates the burst size of the initiating cycle that causes the access violation.
	3:1	RO	0	Indicates the `PROT` of the initiating cycle that causes the access violation.
	0	RO	0	Write access violation log for the transaction is valid only when this bit is set. This bit is cleared when the interrupt register is cleared.
`0x194`	31:0	RO	0	Master ID for the cycle that causes the access violation.
`0x198`	31:0	RO	0	Write target address for the cycle that causes the access violation (lower 32-bit).
`0x19C`	31:0	RO	0	Write target address for the cycle that causes the access violation (upper 32-bit). Valid only if widest address in system is larger than 32 bits.

Offset	Bit	Attribute	Default	Description
`0x1A0`	31:0	R0	0	First 32 bits of the write data for the write cycle that causes the access violation. **Note:** When this register is read, the current write access violation log is recovered from FIFO, when the data width is 32 bits.
`0x1A4`	31:0	R0	0	Bits [`63:32`] of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 32 -bits.
`0x1A8`	31:0	R0	0	Bits [`95:64`] of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits.
`0x1AC`	31:0	R0	0	The first bits (`127:96`) of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits. **Note:** When this register is read, the current write access violation log is recovered from FIFO.

Designating a Default Slave in the System Contents Tab

You can designate any slave in your Qsys system as the error response default slave. The designated default slave provides an error response service for masters that attempt access to an undefined memory region.

1. In your Qsys system, in the **System Contents** tab, right-click the header and turn on **Show Default Slave Column**.
2. Select the slave that you want to designate as the default slave, and then click the checkbox for the slave in the **Default Slave** column.
3. In the **System Contents** tab, in the **Connections** column, connect the designated default slave to one or more masters.

Tri-State Components

The tri-state interface type allows you to design Qsys subsystems that connect to tri-state devices on your PCB. You can use tri-state components to implement pin sharing, convert between unidirectional and bidirectional signals, and create tri-state controllers for devices whose interfaces can be described using the tri-state signal types.

Figure 10-19: Tri-State Conduit System to Control Off-Chip SRAM and Flash Devices

In this example, there are two generic Tri-State Conduit Controllers. The first is customized to control a flash memory. The second is customized to control an off-chip SSRAM. The Tri-State Conduit Pin Sharer multiplexes between these two controllers, and the Tri-State Conduit Bridge converts between an on-chip encoding of tri-state signals and true bidirectional signals. By default, the Tri-State Conduit Pin Sharer and Tri-State Conduit Bridge present byte addresses. Typically, each address location contains more than one byte of data.

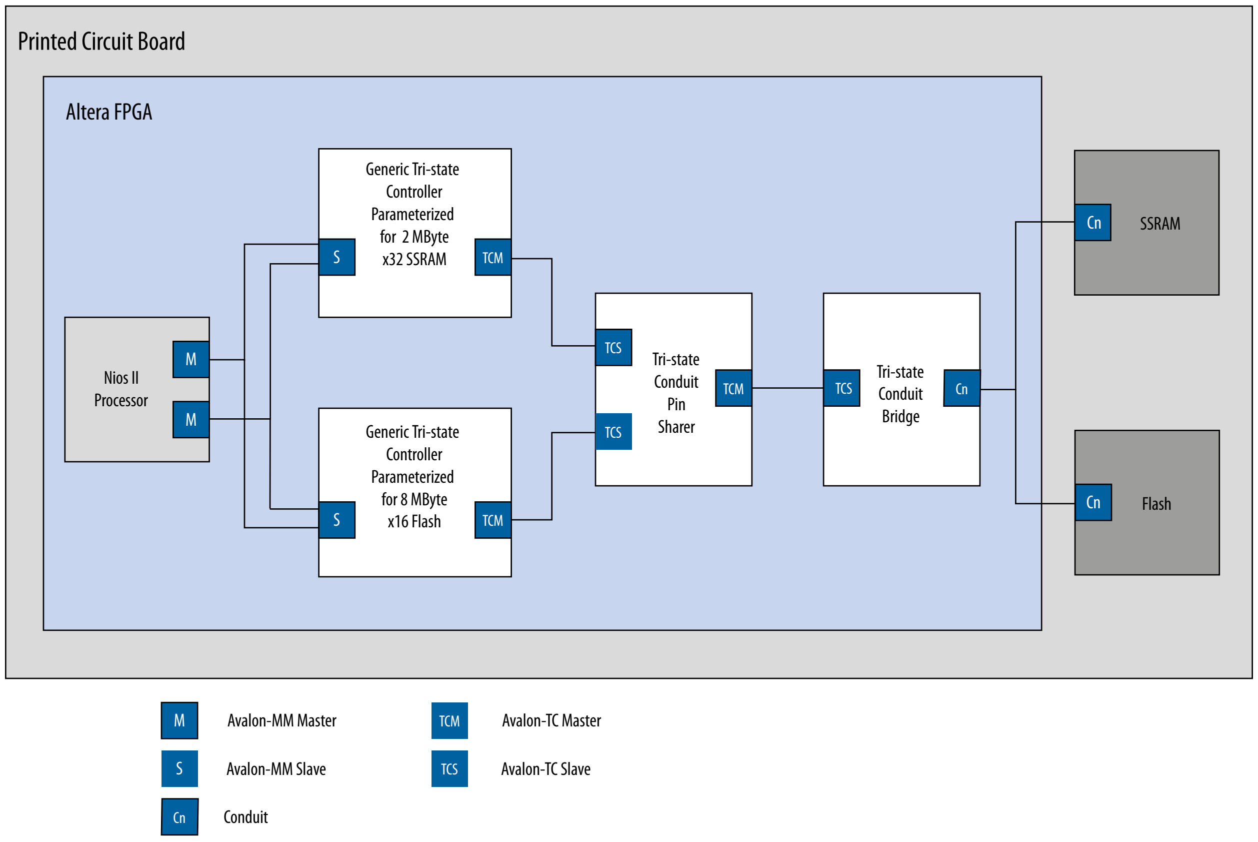
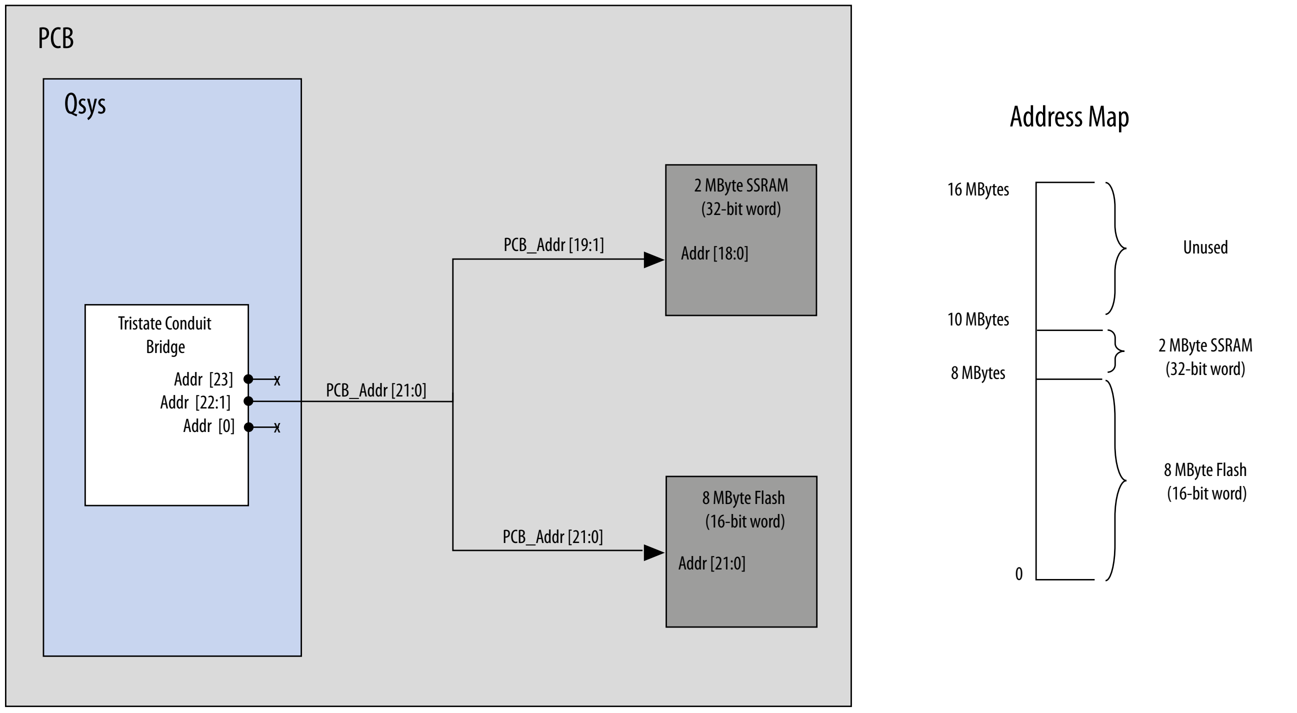


Figure 10-20: Address Connections from Qsys System to PCB

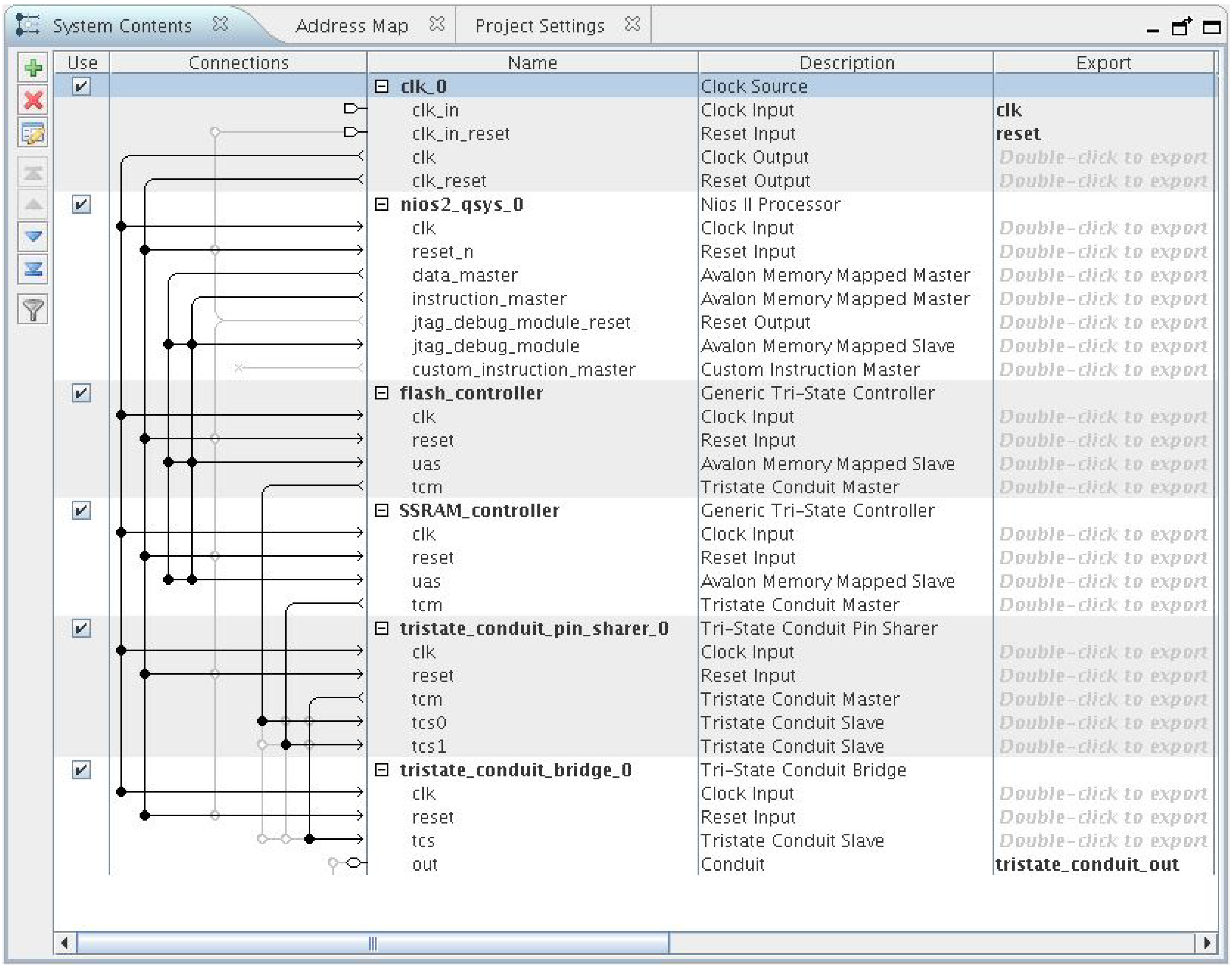
The flash device operates on 16-bit words and must ignore the least-significant bit of the Avalon-MM address, and shows `addr[0]`as not connected. The SSRAM memory operates on 32-bit words and must ignore the two, low-order memory bits. Because neither device requires a byte address, `addr[0]` is not routed on the PCB.

The flash device responds to address range 0 MBytes to 8 MBytes-1. The SSRAM responds to address range 8 MBytes to 10 MBytes-1. The PCB schematic for the PCB connects `addr [21:0]` to `addr [18:0]` of the SSRAM device because the SSRAM responds to 32-bit word address. The 8 MByte flash device accesses 16-bit words; consequently, the schematic does not connect `addr[0]`. The `chipselect` signals select between the two devices.



Note: If you create a custom tri-state conduit master with word aligned addresses, the Tri-state Conduit Pin Sharer does not change or align the address signals.

Figure 10-21: Tri-State Conduit System in Qsys



Use	Connections	Name	Description	Export
✓		⊟ clk_0	Clock Source	
		clk_in	Clock Input	clk
		clk_in_reset	Reset Input	reset
		clk	Clock Output	Double-click to export
		clk_reset	Reset Output	Double-click to export
✓		⊟ nios2_qsys_0	Nios II Processor	
		clk	Clock Input	Double-click to export
		reset_n	Reset Input	Double-click to export
		data_master	Avalon Memory Mapped Master	Double-click to export
		instruction_master	Avalon Memory Mapped Master	Double-click to export
		jtag_debug_module_reset	Reset Output	Double-click to export
		jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export
		custom_instruction_master	Custom Instruction Master	Double-click to export
✓		⊟ flash_controller	Generic Tri-State Controller	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		uas	Avalon Memory Mapped Slave	Double-click to export
		tcm	Tristate Conduit Master	Double-click to export
✓		⊟ SSRAM_controller	Generic Tri-State Controller	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		uas	Avalon Memory Mapped Slave	Double-click to export
		tcm	Tristate Conduit Master	Double-click to export
✓		⊟ tristate_conduit_pin_sharer_0	Tri-State Conduit Pin Sharer	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		tcm	Tristate Conduit Master	Double-click to export
		tcs0	Tristate Conduit Slave	Double-click to export
		tcs1	Tristate Conduit Slave	Double-click to export
✓		⊟ tristate_conduit_bridge_0	Tri-State Conduit Bridge	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		tcs	Tristate Conduit Slave	Double-click to export
		out	Conduit	tristate_conduit_out

Related Information

- Avalon Interface Specifications
- Avalon Tri-State Conduit Components User Guide

Generic Tri-State Controller

The Generic Tri-State Controller provides a template for a controller. You can customize the tri-state controller with various parameters to reflect the behavior of an off-chip device. The following types of parameters are available for the tri-state controller:

- Width of the address and data signals
- Read and write wait times
- Bus-turnaround time
- Data hold time

Note: In calculating delays, the Generic Tri-State Controller chooses the larger of the bus-turnaround time and read latency. Turnaround time is measured from the time that a command is accepted, not from the time that the previous read returned data.

The Generic Tri-State Controller includes the following interfaces:

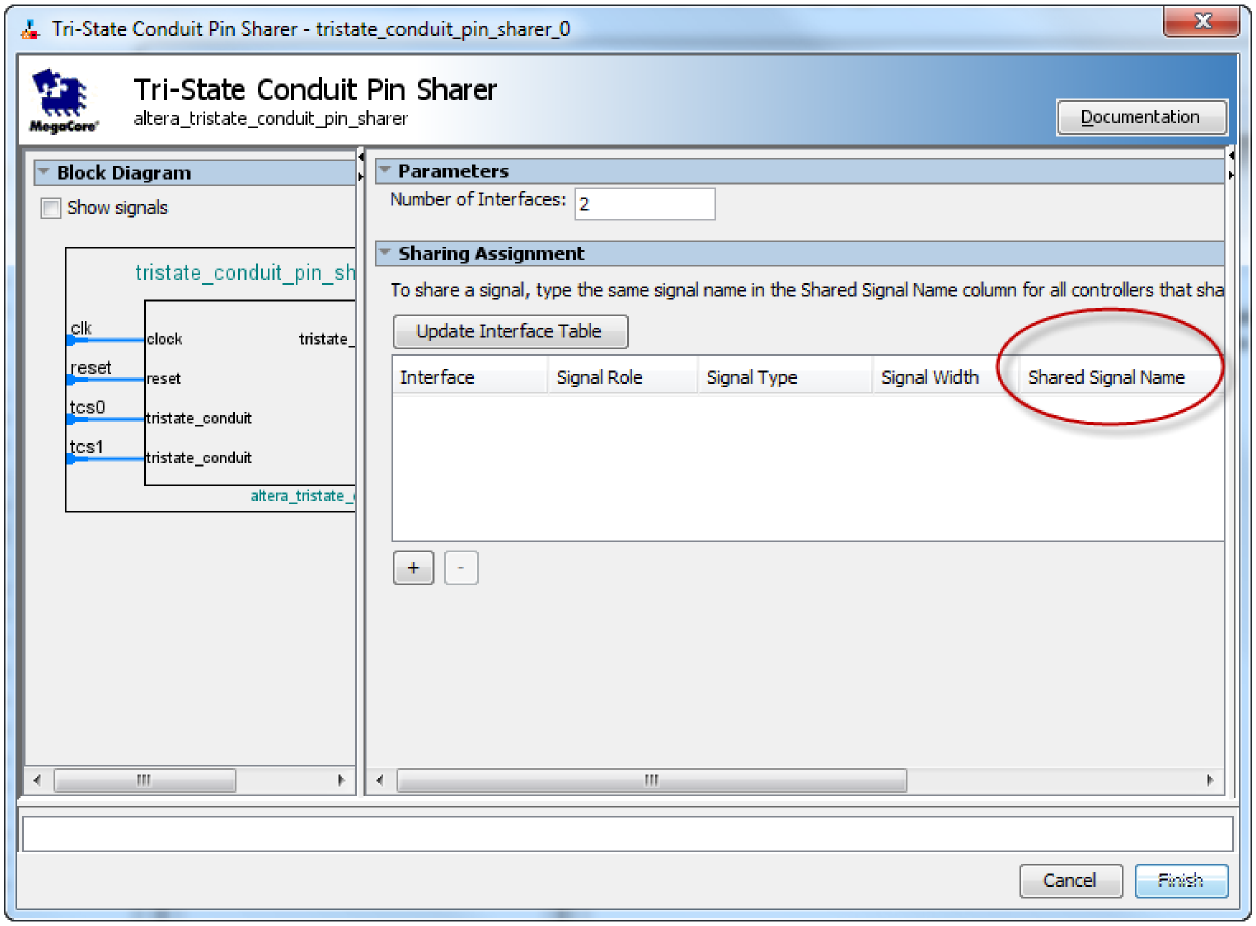
- **Memory-mapped slave interface**—This interface connects to an memory-mapped master, such as a processor.
- **Tristate Conduit Master interface**—Tri-state master interface usually connects to the tri-state conduit slave interface of the tri-state conduit pin sharer.
- **Clock sink**—The component's clock reference. You must connect this interface to a clock source.
- **Reset sink**—This interface connects to a reset source interface.

Tri-State Conduit Pin Sharer

The Tri-state Conduit Pin Sharer multiplexes between the signals of the connected tri-state controllers. You connect all signals from the tri-state controllers to the Tri-state Conduit Pin Sharer and use the parameter editor to specify the signals that are shared.

Figure 10-22: Tri-State Conduit Pin Sharer Parameter Editor

The parameter editor includes a **Shared Signal Name** column. If the widths of shared signals differ, the signals are aligned on their 0th bit and the higher-order pins are driven to 0 whenever the smaller signal has control of the bus. Unshared signals always propagate through the pin sharer. The tri-state conduit pin sharer uses the round-robin arbiter to select between tri-state conduit controllers.



Tri-State Conduit Pin Sharer - tristate_conduit_pin_sharer_0

Tri-State Conduit Pin Sharer
altera_tristate_conduit_pin_sharer

Number of Interfaces: 2

To share a signal, type the same signal name in the Shared Signal Name column for all controllers that sha

Interface	Signal Role	Signal Type	Signal Width	Shared Signal Name

Note: All tri-state conduit components are connected to a pin sharer must be in the same clock domain.

Related Information

Avalon-ST Round Robin Scheduler on page 10-66

Tri-State Conduit Bridge

The Tri-State Conduit Bridge instantiates bidirectional signals for each tri-state signal while passing all other signals straight through the component. The Tri-State Conduit Bridge registers all outgoing and incoming signals, which adds two cycles of latency for a read request. You must account for this additional pipelining when designing a custom controller. During reset, all outputs are placed in a high-impedance state. Outputs are enabled in the first clock cycle after reset is deasserted, and the output signals are then bidirectional.

Test Pattern Generator and Checker Cores

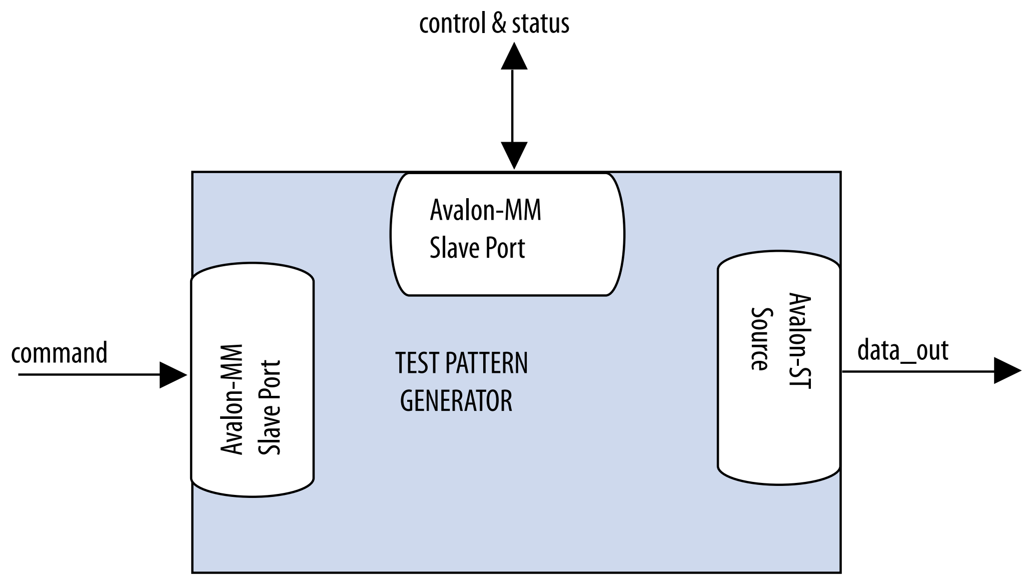
The data generation and monitoring solution for Avalon-ST consists of two components: a test pattern generator core that generates data, and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and verifies it. Optionally, the data can be formatted as packets, with accompanying `start_of_packet` and `end_of_packet` signals.

The test pattern generator inserts different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) slave. The **Throttle Seed** is the starting value for the throttle control random number generator. Altera recommends a unique value for each instance of the test pattern generator and checker cores in a system.

Test Pattern Generator

Figure 10-23: Test Pattern Generator Core

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface, such as the number of error bits and data signal width, thus allowing you to test components with different interfaces.



The data pattern is calculated as: *Symbol Value = Symbol Position in Packet* `XOR` *Data Error Mask*. Data that is not organized in packets is a single stream with no beginning or end. The test pattern generator has a throttle register that is set via the Avalon-MM control interface. The test pattern generator uses the value of the throttle register in conjunction with a pseudo-random number generator to throttle the data generation rate.

Test Pattern Generator Command Interface

The command interface for the Test Pattern Generator is a 32-bit Avalon-MM write slave that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a master peripheral to drive a number of commands into the test pattern generator.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is addressed. When the FIFO is full, the command interface asserts the `waitrequest` signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are cleared when 0 is written to this register, or its respective fields.

Test Pattern Generator Control and Status Interface

The control and status interface of the Test Pattern Generator is a 32-bit Avalon-MM slave that allows you to enable or disable the data generation, as well as set the throttle. This interface also provides generation-time information, such as the number of channels and whether or not data packets are supported.

Test Pattern Generator Output Interface

The output interface of the Test Pattern Generator is an Avalon-ST interface that optionally supports data packets. You can configure the output interface to align with your system requirements. Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator maintains an internal state for each channel.

You can configure the output interface of the test pattern generator with the following parameters:

- **Number of Channels**—Number of channels that the test pattern generator supports. Valid values are 1 to 256.
- **Data Bits Per Symbol**—Bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—Number of symbols (words) that are transferred per beat. Valid values are 1 to 256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—Width of the error signal on the output interface. Valid values are 0 to 31. A value of `0` indicates that the error signal is not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

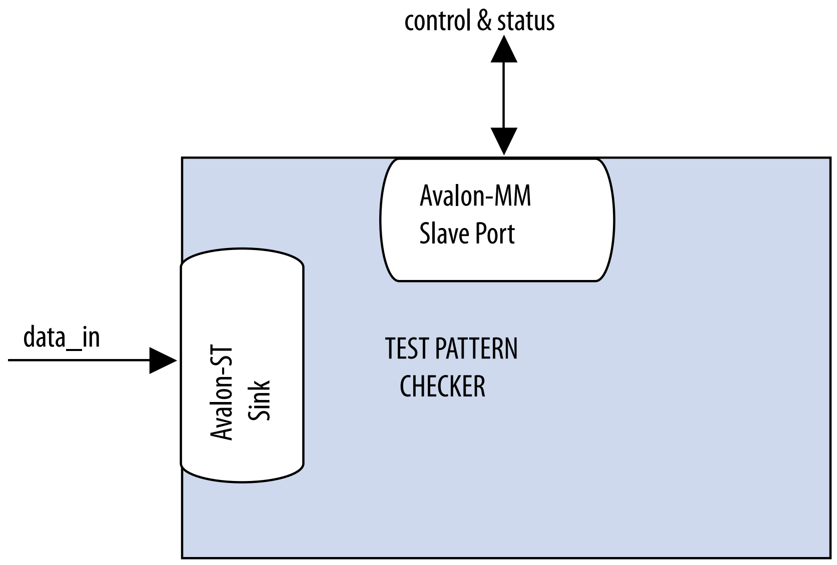
Test Pattern Generator Functional Parameter

The Test Pattern Generator functional parameter allows you to configure the test pattern generator as a whole system.

Test Pattern Checker

Figure 10-24: Test Pattern Checker

The test pattern checker core accepts data via an Avalon-ST interface and verifies it against the same predetermined pattern that the test pattern generator uses to produce the data. The test pattern checker core reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width. This enables the ability to test components with different interfaces. The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.



The test pattern checker detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP), and signaled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Test Pattern Checker Input Interface

The Test Pattern Checker input interface is an Avalon-ST interface that optionally supports data packets. You can configure the input interface to align with your system requirements. Incoming data may contain interleaved packet fragments. To keep track of the current symbol's position, the test pattern checker maintains an internal state for each channel.

Test Pattern Checker Control and Status Interface

The Test Pattern Checker control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable data acceptance, as well as set the throttle. This interface provides generation-time information, such as the number of channels and whether the test pattern checker supports data packets. The control and status interface also provides information on the exceptions detected by the test pattern checker. The interface obtains this information by reading from the exception FIFO.

Test Pattern Checker Functional Parameter

The Test Pattern Checker functional parameter allows you to configure the test pattern checker as a whole system.

Test Pattern Checker Input Parameters

You can configure the input interface of the test pattern checker using the following parameters:

- **Data Bits Per Symbol**—Bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—Number of symbols (words) that are transferred per beat. Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Number of Channels**—Number of channels that the test pattern checker supports. Valid values are 1 to 256.
- **Error Signal Width (bits)**—Width of the `error` signal on the input interface. Valid values are 0 to 31. A value of `0` indicates that the `error` signal in not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Software Programming Model for the Test Pattern Generator and Checker Cores

The HAL system library support, software files, and register maps describe the software programming model for the test pattern generator and checker cores.

HAL System Library Support

For Nios II processor users, Altera provides HAL system library drivers that allow you to initialize and access the test pattern generator and checker cores. Altera recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- ***<IP installation directory>*/ip/sopc_builder_ip/altera_avalon_data_source/HAL**
- ***<IP installation directory>*/ip/sopc_builder_ip/altera_avalon_data_sink/HAL**

Note: This instruction does not apply if you use the Nios II command-line tools.

Test Pattern Generator and Test Pattern Checker Core Files

The following files define the low-level access to the hardware, and provide the routines for the HAL device drivers.

Note: Do not modify the test pattern generator or test pattern checker core files.

- Test pattern generator core files:
  - **data_source_regs.h**—Header file that defines the test pattern generator's register maps.
  - **data_source_util.h , data_source_util.c**—Header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Test pattern checker core files:
  - **data_sink_regs.h**—Header file that defines the core's register maps.
  - **data_sink_util.h , data_sink_util.c**—Header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps for the Test Pattern Generator and Test Pattern Checker Cores

Test Pattern Generator Control and Status Registers

Table 10-13: Test Pattern Generator Control and Status Register Map

Shows the offset for the test pattern generator control and status registers. Each register is 32-bits wide.

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	`fill`

Table 10-14: Test Pattern Generator Status Register Bits

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	A constant value of `0x64`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates data packet support.

Table 10-15: Test Pattern Generator Control Register Bits

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern generator core.
[7:1]	Reserved		

Bit(s)	Name	Access	Description
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. The test pattern generator uses this value in conjunction with a pseudo-random number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 10-16: Test Pattern Generator Fill Register Bits

Bit(s)	Name	Access	Description
[0]	`BUSY`	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
[6:1]	Reserved		
[15:7]	`FILL`	RO	The number of commands currently in the command FIFO.
[31:16]	Reserved		

Test Pattern Generator Command Registers

Table 10-17: Test Pattern Generator Command Register Map

Shows the offset for the command registers. Each register is 32-bits wide.

Offset	Register Name
base + 0	`cmd_lo`
base + 1	`cmd_hi`

The `cmd_lo` is pushed into the FIFO only when the `cmd_lo` register is addressed.

Table 10-18: `cmd_lo` Register Bits

Bit(s)	Name	Access	Description
[15:0]	`SIZE`	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
[29:16]	`CHANNEL`	RW	The channel to send the segment on. If the `channel` signal is less than 14 bits wide, the test pattern generator uses the low order bits of this register to drive the signal.
[30]	`SOP`	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when data packets are not supported.
[31]	`EOP`	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when data packets are not supported.

Table 10-19: `cmd_hi` Register Bits

Bit(s)	Name	Access	Description
[15:0]	`SIGNALED ERROR`	RW	Specifies the value to drive the `error` signal. A non-zero value creates a signalled error.
[23:16]	`DATA ERROR`	RW	The output data is `XOR`ed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
[24]	`SUPPRESS SOP`	RW	Set this bit to 1 to suppress the assertion of the `startofpacket` signal when the first segment in a packet is sent.
[25]	`SUPRESS EOP`	RW	Set this bit to 1 to suppress the assertion of the `endofpacket` signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

Table 10-20: Test Pattern Checker Control and Status Register Map

Shows the offset for the control and status registers. Each register is 32 bits wide.

Offset	Register Name
base + 0	`status`
base + 1	`control`

Offset	Register Name
base + 2	Reserved
base + 3	
base + 4	
base + 5	`exception_descriptor`
base + 6	`indirect_select`
base + 7	`indirect_count`

Table 10-21: Test Pattern Checker Status Register Bits

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	Contains a constant value of `0x65`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 10-22: Test Pattern Checker Control Register Bits

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern checker.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. Qsys uses this value in conjunction with a pseudo-random number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

If there is no exception, reading the `exception_descriptor` register bit register returns 0.

Table 10-23: `exception_descriptor` Register Bits

Bit(s)	Name	Access	Description
[0]	`DATA ERROR`	RO	A value of 1 indicates that an error is detected in the incoming data.
[1]	`MISSINGSOP`	RO	A value of 1 indicates missing start-of-packet.
[2]	`MISSINGEOP`	RO	A value of 1 indicates missing end-of-packet.
[7:3]	Reserved		
[15:8]	`SIGNALLED ERROR`	RO	The value of the `error` signal.
[23:16]	Reserved		
[31:24]	`CHANNEL`	RO	The channel on which the exception was detected.

Table 10-24: `indirect_select` Register Bits

Bit	Bits Name	Access	Description
[7:0]	`INDIRECT CHANNEL`	RW	Specifies the channel number that applies to the `INDIRECT PACKET COUNT`, `INDIRECT SYMBOL COUNT`, and `INDIRECT ERROR COUNT` registers.
[15:8]	Reserved		
[31:16]	`INDIRECT ERROR`	RO	The number of data errors that occurred on the channel specified by `INDIRECT CHANNEL`.

Table 10-25: `indirect_count` Register Bits

Bit	Bits Name	Access	Description
[15:0]	`INDIRECT PACKET COUNT`	RO	The number of data packets received on the channel specified by `INDIRECT CHANNEL`.
[31:16]	`INDIRECT SYMBOL COUNT`	RO	The number of symbols received on the channel specified by `INDIRECT CHANNEL`.

.

Test Pattern Generator API

The following subsections describe application programming interface (API) for the test pattern generator.

Note: API functions are currently not available from the interrupt service routine (ISR).

data_source_reset()

Table 10-26: data_source_reset()

Information Type	Description
Prototype	`void data_source_reset(alt_u32 base);`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`void`
Description	Resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

Table 10-27: data_source_init()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base, alt_u32 command_base);`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave. `command_base`—Base address of the command slave.
Returns	`1`—Initialization is successful. `0`—Initialization is unsuccessful.
Description	Performs the following operations to initialize the test pattern generator core: • Resets and disables the test pattern generator core. • Sets the maximum throttle. • Clears all inserted errors.

data_source_get_id()

Table 10-28: data_source_get_id()

Information Type	Description
Prototype	`int data_source_get_id(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Test pattern generator core identifier.
Description	Retrieves the test pattern generator core's identifier.

data_source_get_supports_packets()

Table 10-29: data_source_get_supports_packets()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Data packets are supported. `0`—Data packets are not supported.
Description	Checks if the test pattern generator core supports data packets.

data_source_get_num_channels()

Table 10-30: data_source_get_num_channels()

Description	Description
Prototype	`int data_source_get_num_channels(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of channels supported.
Description	Retrieves the number of channels supported by the test pattern generator core.

data_source_get_symbols_per_cycle()

Table 10-31: data_source_get_symbols_per_cycle()

Description	Description
Prototype	`int data_source_get_symbols(alt_u32 base);`
Thread-safe	Yes

Description	Description
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of symbols transferred in a beat.
Description	Retrieves the number of symbols transferred by the test pattern generator core in each beat.

data_source_get_enable()

Table 10-32: data_source_get_enable()

Information Type	Description
Prototype	`int data_source_get_enable(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Value of the `ENABLE` bit.
Description	Retrieves the value of the `ENABLE` bit.

data_source_set_enable()

Table 10-33: data_source_set_enable()

Information Type	Description
Prototype	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave. `value`— `ENABLE` bit set to the value of this parameter.
Returns	`void`

Information Type	Description
Description	Enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO

data_source_get_throttle()

Table 10-34: data_source_get_throttle()

Information Type	Description
Prototype	`int data_source_get_throttle(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Throttle value.
Description	Retrieves the current throttle value.

data_source_set_throttle()

Table 10-35: data_source_set_throttle()

Information Type	Description
Prototype	`void data_source_set_throttle(alt_u32 base, alt_u32 value) ;`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave. `value`—Throttle value.
Returns	`void`
Description	Sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

data_source_is_busy()

Table 10-36: data_source_is_busy()

Information Type	Description
Prototype	`int data_source_is_busy(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Test pattern generator core is busy. `0`—Test pattern generator core is not busy.
Description	Checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

data_source_fill_level()

Table 10-37: data_source_fill_level()

Information Type	Description
Prototype	`int data_source_fill_level(alt_u32 base);`
Thread-safe	Yes
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of commands in the command FIFO.
Description	Retrieves the number of commands currently in the command FIFO.

data_source_send_data()

Table 10-38: data_source_send_data()

Information Type	Description
Prototype	`int data_source_send_data(alt_u32 cmd_base, alt_u16 channel, alt_u16 size, alt_u32 flags, alt_u16 error, alt_u8 data_error_mask);`

Information Type	Description
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`cmd_base`—Base address of the command slave. `channel`—Channel to send the data. `size`—Data size. `flags` —Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`. `error`—Value asserted on the `error` signal on the output interface. `data_error_mask`—Parameter and the data are `XOR`ed together to produce erroneous data.
Returns	Returns `1`.
Description	Sends a data fragment to the specified channel. If data packets are supported, applications must ensure consistent usage of SOP and EOP in each channel. Except for the last segment in a packet, the length of each segment is a multiple of the data width. If data packets are not supported, applications must ensure that there are no SOP and EOP indicators in the data. The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

The following subsections describe API for the test pattern checker core. The API functions are currently not available from the ISR.

data_sink_reset() on page 10-53

data_sink_init() on page 10-53

data_sink_get_id() on page 10-54

data_sink_get_supports_packets() on page 10-54

data_sink_get_num_channels() on page 10-55

data_sink_get_symbols_per_cycle() on page 10-55

data_sink_get_enable() on page 10-55

data_sink_set enable() on page 10-56

data_sink_get_throttle() on page 10-56

data_sink_set_throttle() on page 10-57

data_sink_reset()

Table 10-39: data_sink_reset()

Information Type	Description
Prototype	`void data_sink_reset(alt_u32 base);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`void`
Description	Resets the test pattern checker core including all internal counters.

data_sink_init()

Table 10-40: data_sink_init()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.

Information Type	Description
Returns	`1`—Initialization is successful. `0`—Initialization is unsuccessful.
Description	Performs the following operations to initialize the test pattern checker core: • Resets and disables the test pattern checker core. • Sets the throttle to the maximum value.

data_sink_get_id()

Table 10-41: data_sink_get_id()

Information Type	Description
Prototype	`int data_sink_get_id(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Test pattern checker core identifier.
Description	Retrieves the test pattern checker core's identifier.

data_sink_get_supports_packets()

Table 10-42: data_sink_get_supports_packets()

Information Type	Description
Prototype	`int data_sink_init(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Data packets are supported. `0`—Data packets are not supported.
Description	Checks if the test pattern checker core supports data packets.

data_sink_get_num_channels()

Table 10-43: data_sink_get_num_channels()

Information Type	Description
Prototype	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of channels supported.
Description	Retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

Table 10-44: data_sink_get_symbols_per_cycle()

Information Type	Description
Prototype	`int data_sink_get_symbols(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of symbols received in a beat.
Description	Retrieves the number of symbols received by the test pattern checker core in each beat.

data_sink_get_enable()

Table 10-45: data_sink_get_enable()

Information Type	Description
Prototype	`int data_sink_get_enable(alt_u32 base);`
Thread-safe	Yes

Information Type	Description
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Value of the `ENABLE` bit.
Description	Retrieves the value of the `ENABLE` bit.

data_sink_set enable()

Table 10-46: data_sink_set enable()

Information Type	Description
Prototype	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave. `value`—`ENABLE` bit is set to the value of the parameter.
Returns	`void`
Description	Enables the test pattern checker core.

data_sink_get_throttle()

Table 10-47: data_sink_get_throttle()

Information Type	Description
Prototype	`int data_sink_get_throttle(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Throttle value.
Description	Retrieves the throttle value.

data_sink_set_throttle()

Table 10-48: data_sink_set_throttle()

Information Type	Description
Prototype	`void data_sink_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe	No
Include:	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave. `value`—Throttle value.
Returns	`void`
Description	Sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

data_sink_get_packet_count()

Table 10-49: data_sink_get_packet_count()

Information Type	Description
Prototype	`int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of data packets received on the channel.
Description	Retrieves the number of data packets received on a channel.

data_sink_get_error_count()

Table 10-50: data_sink_get_error_count()

Information Type	Description
Prototype	`int data_sink_get_error_count(alt_u32 base, alt_u32 channel);`
Thread-safe	No
Include	<***data_sink_util.h***>
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of errors received on the channel.
Description	Retrieves the number of errors received on a channel.

data_sink_get_symbol_count()

Table 10-51: data_sink_get_symbol_count()

Information Type	Description
Prototype	`int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`
Thread-safe	No
Include	<***data_sink_util.h***>
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of symbols received on the channel.
Description	Retrieves the number of symbols received on a channel.

data_sink_get_exception()

Table 10-52: data_sink_get_exception()

Information Type	Description
Prototype	`int data_sink_get_exception(alt_u32 base);`

Information Type	Description
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	First exception descriptor in the exception FIFO. `0`—No exception descriptor found in the exception FIFO.
Description	Retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

data_sink_exception_is_exception()

Table 10-53: data_sink_exception_is_exception()

Information Type	Description
Prototype	`int data_sink_exception_is_exception(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor
Returns	`1`—Indicates an exception. `0`—No exception.
Description	Checks if an exception descriptor describes a valid exception.

data_sink_exception_has_data_error()

Table 10-54: data_sink_exception_has_data_error()

Information Type	Description
Prototype	`int data_sink_exception_has_data_error(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.

Information Type	Description
Returns	`1`—Data has errors. `0`—No errors.
Description	Checks if an exception indicates erroneous data.

data_sink_exception_has_missing_sop()

Table 10-55: data_sink_exception_has_missing_sop()

Information Type	Description
Prototype	`int data_sink_exception_has_missing_sop(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	`1`—Missing SOP. `0`—Other exception types.
Description	Checks if an exception descriptor indicates missing SOP.

data_sink_exception_has_missing_eop()

Table 10-56: data_sink_exception_has_missing_eop()

Information Type	Description
Prototype	`int data_sink_exception_has_missing_eop(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	`1`—Missing EOP. `0`—Other exception types.
Description	Checks if an exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Table 10-57: data_sink_exception_signalled_error()

Information Type	Description
Prototype	`int data_sink_exception_signalled_error(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	Signal error value.
Description	Retrieves the value of the signaled error from the exception.

data_sink_exception_channel()

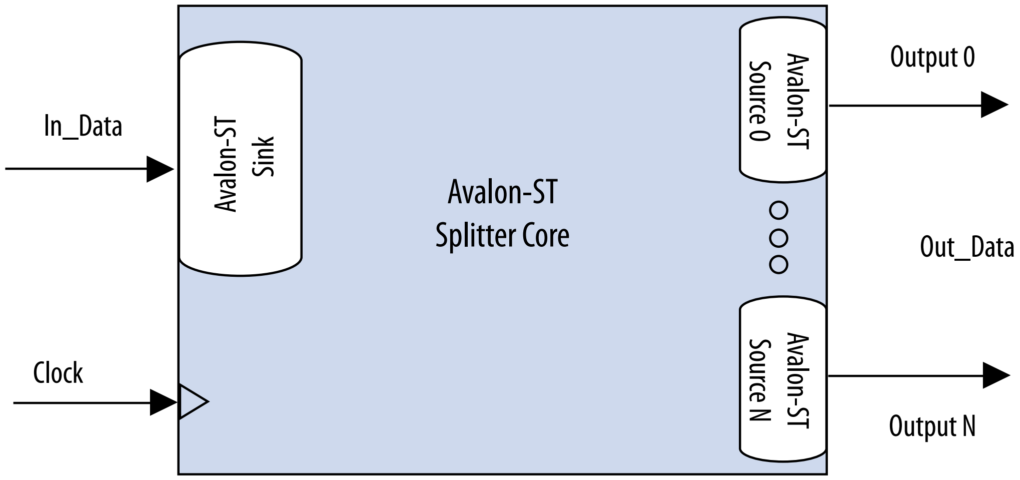
Table 10-58: data_sink_exception_channel()

Information Type	Description
Prototype	`int data_sink_exception_channel(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	Channel number on which an exception occurred.
Description	Retrieves the channel number on which an exception occurred.

Avalon-ST Splitter Core

Figure 10-25: Avalon-ST Splitter Core

The Avalon-ST Splitter Core allows you to replicate transactions from an Avalon-ST source interface to multiple Avalon-ST sink interfaces. This core supports from 1 to 16 outputs.



The Avalon-ST Splitter core copies input signals from the input interface to the corresponding output signals of each output interface without altering the size or functionality. This includes all signals except for the `ready` signal. The core includes a clock signal to determine the Avalon-ST interface and clock domain where the core resides. Because the splitter core does nor use the `clock` signal internally, latency is not introduced when using this core.

Splitter Core Backpressure

The Avalon-ST Splitter core integrates with backpressure by AND-ing the `ready` signals from the output interfaces and sending the result to the input interface. As a result, if an output interface deasserts the `ready` signal, the input interface receives the deasserted `ready` signal, as well. This functionality ensures that backpressure on the output interfaces is propagated to the input interface.

When the **Qualify Valid Out** parameter is set to 1, the `out_valid` signals on all other output interfaces are gated when backpressure is applied from one output interface. In this case, when any output interface deasserts its `ready` signal, the `out_valid` signals on the other output interfaces are also deasserted.

When the **Qualify Valid Out** parameter is set to 0, the output interfaces have a non-gated `out_valid` signal when backpressure is applied. In this case, when an output interface deasserts its ready signal, the `out_valid` signals on the other output interfaces are not affected.

Because the logic is combinational, the core introduces no latency.

Splitter Core Interfaces

The Avalon-ST Splitter core supports streaming data, with optional packet, channel, and error signals. The core propagates backpressure from any output interface to the input interface.

Table 10-59: Avalon-ST Splitter Core Support

Feature	Support
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Splitter Core Parameters

Table 10-60: Avalon-ST Splitter Core Parameters

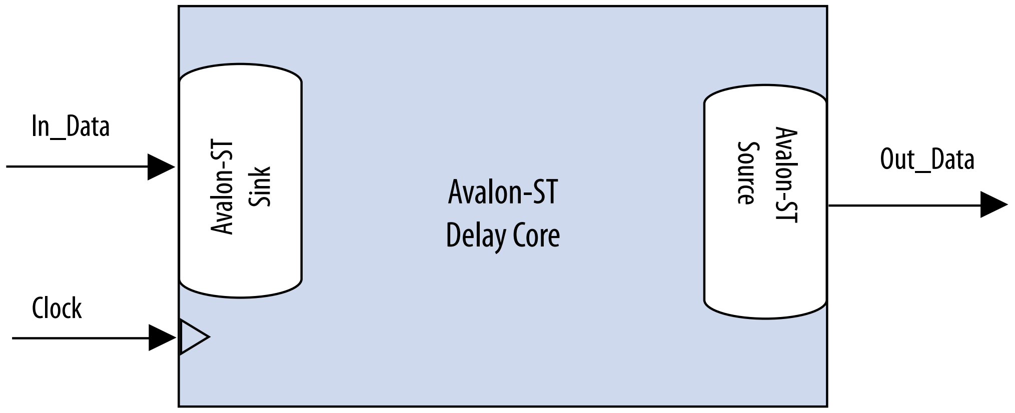
Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. Qsys supports 1 for some systems where no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `out_valid` signal is gated or non-gated when backpressure is applied.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.

Parameter	Legal Values	Default Value	Description
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of `0` indicates that the splitter core is not using the `error` signal. This parameter is disabled when **Use Error** is set to `0`.

Avalon-ST Delay Core

Figure 10-26: Avalon-ST Delay Core

The Avalon-ST Delay Core provides a solution to delay Avalon-ST transactions by a constant number of clock cycles. This core supports up to 16 clock cycle delays.



The Delay core adds a delay between the input and output interfaces. The core accepts transactions presented on the input interface and reproduces them on the output interface `N` cycles later without changing the transaction.

The input interface delays the input signals by a constant `N` number of clock cycles to the corresponding output signals of the output interface. The **Number Of Delay Clocks** parameter defines the constant `N`, which must be between 0 and 16. The change of the `in_valid` signal is reflected on the `out_valid` signal exactly `N` cycles later.

Delay Core Reset Signal

The Avalon-ST Delay core has a `reset` signal that is synchronous to the `clk` signal. When the core asserts the `reset` signal, the output signals are held at 0. After the `reset` signal is deasserted, the output signals are held at 0 for `N` clock cycles. The delayed values of the input signals are then reflected at the output signals after `N` clock cycles.

Delay Core Interfaces

The Delay core supports streaming data, with optional packet, channel, and error signals. The delay core does not support backpressure.

Table 10-61: Avalon-ST Delay Core Support

Feature	Support
Backpressure	Not supported.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Delay Core Parameters

Table 10-62: Avalon-ST Delay Core Parameters

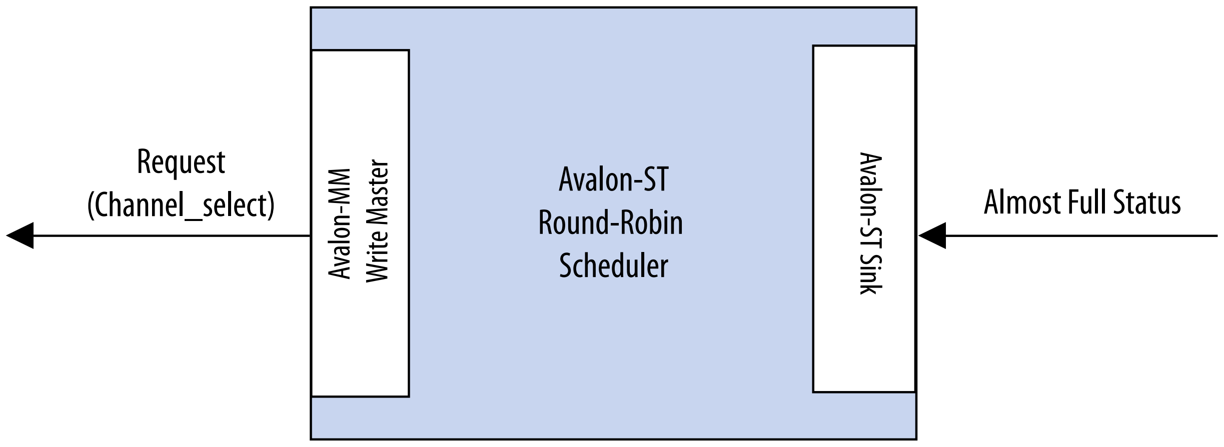
Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. Qsys supports 0 for some systems where no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.

Parameter	Legal Values	Default Value	Description
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not in use. This parameter is disabled when **Use Error** is set to 0.

Avalon-ST Round Robin Scheduler

Figure 10-27: Avalon-ST Round Robin Scheduler

The Avalon-ST Round Robin Scheduler core controls the read operations from a multi-channel Avalon-ST component that buffers data by channels. It reads the almost-full threshold values from the multiple channels in the multi-channel component and issues the read request to the Avalon-ST source according to a round-robin scheduling algorithm.



In a multi-channel component, the component can store data either in the sequence that it comes in (FIFO), or in segments according to the channel. When data is stored in segments according to channels, a scheduler is needed to schedule the read operations.

Almost-Full Status Interface (Round Robin Scheduler)

The Almost-Full Status interface is an Avalon-ST sink interface that collects the almost-full status from the sink components for the channels in the sequence provided.

Table 10-63: Avalon-ST Interface Feature Support

Feature	Property
Backpressure	Not supported
Data Width	Data width = 1; Bits per symbol = 1
Channel	Maximum channel = 32; Channel width = 5
Error	Not supported
Packet	Not supported

Request Interface (Round Robin Scheduler)

The Request Interface is an Avalon-MM write master interface that requests data from a specific channel. The Avalon-ST Round Robin Scheduler cycles through the channels it supports and schedules data to be read.

Round Robin Scheduler Operation

If a particular channel is almost full, the Avalon-ST Round Robin Scheduler does not schedule data to be read from that channel in the source component.

The scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel *n*, the scheduler writes the value `1` to address $(4 \times n)$. For example, if the scheduler is requesting data from channel 3, the scheduler writes `1` to address `0xC`. At every clock cycle, the scheduler requests data from the next channel. Therefore, if the scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, the scheduler uses one clock cycle without a request transaction.

The Avalon-ST Round Robin Scheduler cannot determine if the requested component is able to service the request transaction. The component asserts `waitrequest` when it cannot accept new requests.

Table 10-64: Avalon-ST Round Robin Scheduler Ports

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.

Signal	Direction	Description
Avalon-MM Request Interface		
`request_address (log`$_2$` Max_Channels-1:0)`	Out	The write address that indicates which channel has the request.
`request_write`	Out	Write enable signal.
`request_writedata`	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
`request_waitrequest`	In	Wait request signal that pauses the scheduler when the slave cannot accept a new request.
Avalon-ST Almost-Full Status Interface		
`almost_full_valid`	In	Indicates that `almost_full_channel` and `almost_full_data` are valid.
`almost_full_channel (Channel_Width-1:0)`	In	Indicates the channel for the current status indication.
`almost_full_data (log`$_2$` Max_Channels-1:0)`	In	A 1-bit signal that is asserted high to indicate that the channel indicated by `almost_full_channel` is almost full.

Round Robin Scheduler Parameters

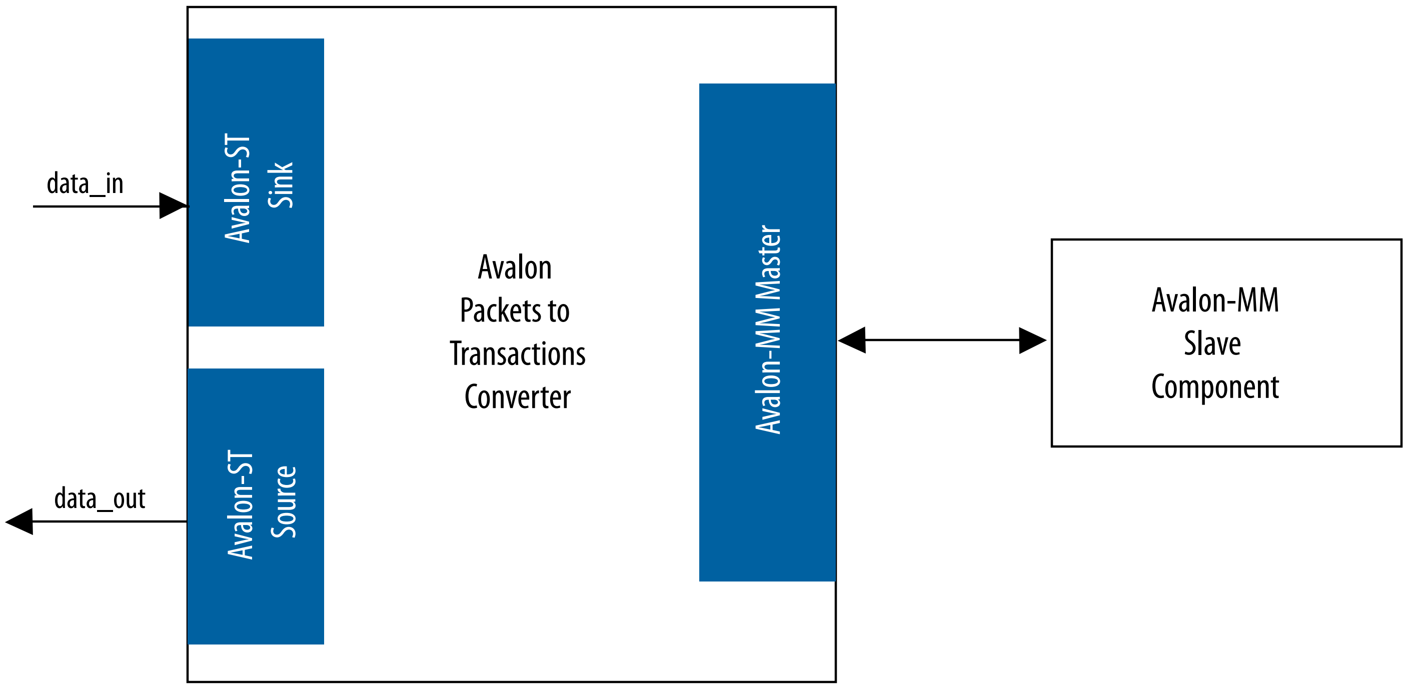
Table 10-65: Avalon-ST Round Robin Scheduler Parameters

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.
Use almost-full status	0–1	Specifies whether the scheduler uses the almost-full interface. If not, the core requests data from the next channel at the next clock cycle.

Avalon Packets to Transactions Converter

Figure 10-28: Avalon Packets to Transactions Converter Core

The Avalon Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon-MM transactions. The core then returns Avalon-MM transaction responses to the requesting components.


data_in
Avalon-ST Sink
Avalon Packets to Transactions Converter
Avalon-MM Master
Avalon-MM Slave Component
data_out
Avalon-ST Source

Note: The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of the Packets to Transactions Converter core. For more information, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

Packets to Transactions Converter Interfaces

Table 10-66: Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.

Feature	Property
Error	Not used.
Packet	Supported.

The Avalon-MM master interface supports read and write transactions. The data width is set to 32 bits, and burst transactions are not supported.

Packets to Transactions Converter Operation

The Packets to Transactions Converter core receives streams of packets on its Avalon-ST sink interface and initiates Avalon-MM transactions. Upon receiving transaction responses from Avalon-MM slaves, the core transforms the responses to packets and returns them to the requesting components via its Avalon-ST source interface. The core does not report Avalon-ST errors.

Packets to Transactions Converter Data Packet Formats

A response packet is returned for every write transaction. The core also returns a response packet if a no transaction (`0x7f`) is received. An invalid transaction code is regarded as a no transaction. For read transactions, the core returns the data read.

The Packets to Transactions Converter core expects incoming data streams to be in the formats shown the table below.

Table 10-67: Data Packet Formats

Byte	Field	Description
Transaction Packet Format		
0	`Transaction code`	Type of transaction.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Transaction size in bytes. For write transactions, the size indicates the size of the `data` field. For read transactions, the size indicates the total number of bytes to read.
[7:4]	`Address`	32-bit address for the transaction.
[n:8]	`Data`	Transaction data; data to be written for write transactions.
Response Packet Format		
0	`Transaction code`	The transaction code with the most significant bit inversed.
1	Reserved	Reserved for future use.

Byte	Field	Description
[4:2]	`Size`	Total number of bytes read/written successfully.

Related Information

- **Packets to Transactions Converter Interfaces** on page 10-69
- **Packets to Transactions Converter Interfaces** on page 10-69

Packets to Transactions Converter Supported Transactions

Table 10-68: Packets to Transactions Converter Supported Transactions

Avalon-MM transactions supported by the Packets to Transactions Converter core.

Transaction Code	Avalon-MM Transaction	Description
`0x00`	Write, non-incrementing address.	Writes data to the address until the total number of bytes written to the same word address equals to the value specified in the `size` field.
`0x04`	Write, incrementing address.	Writes transaction data starting at the current address.
`0x10`	Read, non-incrementing address.	Reads 32 bits of data from the address until the total number of bytes read from the same address equals to the value specified in the `size` field.
`0x14`	Read, incrementing address.	Reads the number of bytes specified in the `size` parameter starting from the current address.
`0x7f`	No transaction.	No transaction is initiated. You can use this transaction type for testing purposes. Although no transaction is initiated on the Avalon-MM interface, the core still returns a response packet for this transaction code.

The Packets to Transactions Converter core can process only a single transaction at a time. The `ready` signal on the core's Avalon-ST sink interface is asserted only when the current transaction is completely processed.

No internal buffer is implemented on the data paths. Data received on the Avalon-ST interface is forwarded directly to the Avalon-MM interface and vice-versa. Asserting the `waitrequest` signal on the Avalon-MM interface backpressures the Avalon-ST sink interface. In the opposite direction, if the Avalon-ST source interface is backpressured, the `read` signal on the Avalon-MM interface is not asserted until the backpressure is alleviated. Backpressuring the Avalon-ST source in the middle of a read could result in data loss. In this cases, the core returns the data that is successfully received.

A transaction is considered complete when the core receives an EOP. For write transactions, the actual data size is expected to be the same as the value of the `size` property. Whether or not both values agree, the core always uses the end of packet (EOP) to determine the end of data.

Packets to Transactions Converter Malformed Packets

The following are examples of malformed packets:

- **Consecutive start of packet (SOP)**—An SOP marks the beginning of a transaction. If an SOP is received in the middle of a transaction, the core drops the current transaction without returning a response packet for the transaction, and initiates a new transaction. This effectively precesses packets without an end of packet (EOP).
- **Unsupported transaction codes**—The core processes unsupported transactions as a no transaction.

Avalon-ST Streaming Pipeline Stage

The Avalon-ST pipeline stage receives data from an Avalon-ST source interface, and outputs the data to an Avalon-ST sink interface. In the absence of back pressure, the Avalon-ST pipeline stage source interface outputs data one cycle after receiving the data on its sink interface.

If the pipeline stage receives back pressure on its source interface, it continues to assert its source interface's current data output. While the pipeline stage is receiving back pressure on its source interface and it receives new data on its sink interface, the pipeline stage internally buffers the new data. It then asserts back pressure on its sink interface.

Once the backpressure is deasserted, the pipeline stage's source interface is de-asserted and the pipeline stage asserts internally buffered data (if present). Additionally, the pipeline stage de-asserts back pressure on its sink interface.

Figure 10-29: Pipeline Stage Simple Register

If the ready signal is not pipelined, the pipeline stage becomes a simple register.

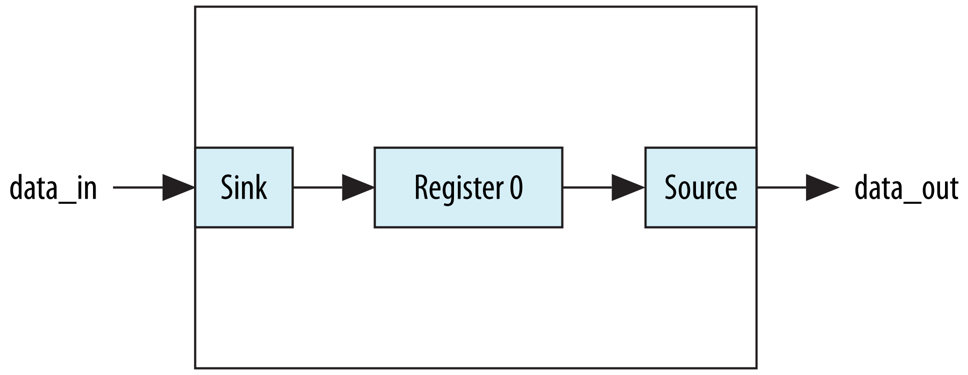
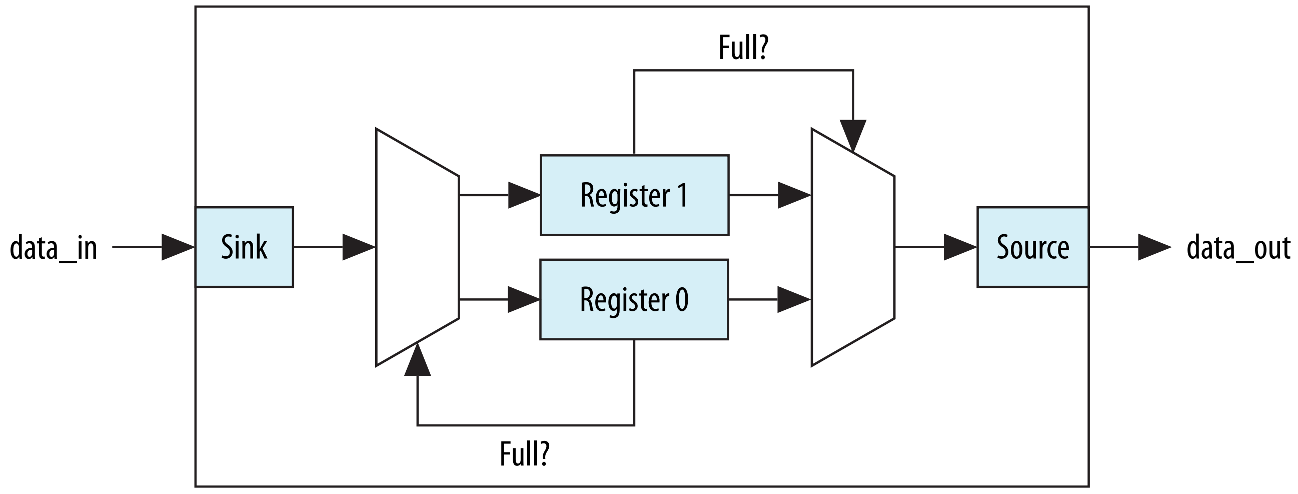


Figure 10-30: Pipeline Stage Holding Register

If the ready signal is pipelined, the pipeline stage must also include a second "holding" register.



Streaming Channel Multiplexer and Demultiplexer Cores

The Avalon-ST channel multiplexer core receives data from various input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate the origin of the data. The Avalon-ST channel demultiplexer core receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer cores can transfer data between interfaces on cores that support unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or demultiplexed data paths without having to write custom HDL code. The multiplexer includes an Avalon-ST Round Robin Scheduler.

Related Information

Avalon-ST Round Robin Scheduler on page 10-66

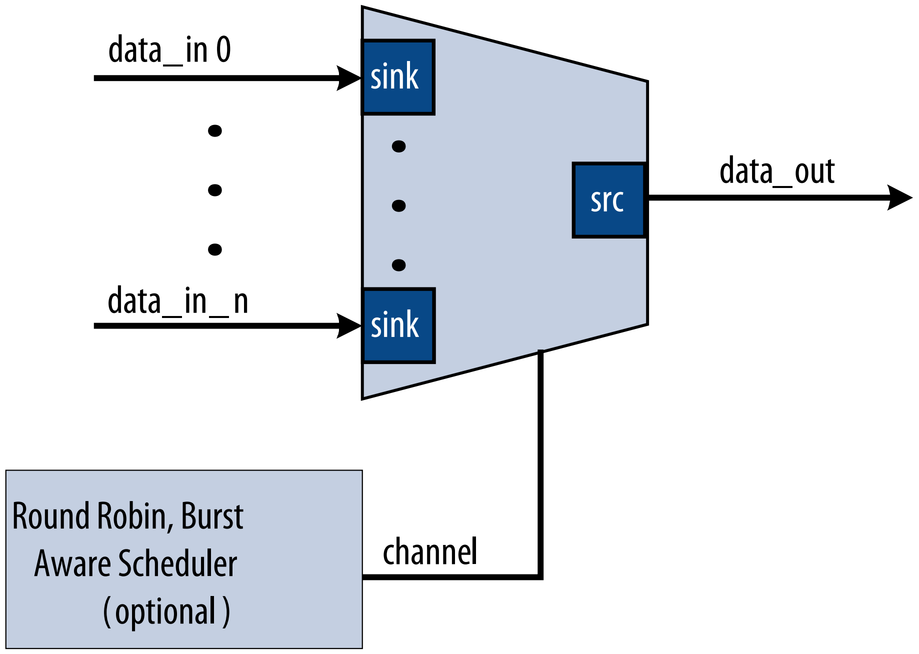
Software Programming Model For the Multiplexer and Demultiplexer Components

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore, Qsys cannot control or configure any aspect of the multiplexer or demultiplexer at run-time. The components cannot generate interrupts.

Avalon-ST Multiplexer

Figure 10-31: Avalon-ST Multiplexer

The Avalon-ST multiplexer takes data from a variety of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that the other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.



The multiplexer includes an optional channel signal that enables each input interface to carry channelized data. The output interface channel width is equal to:

(log2 (n-1)) + 1 + w

where `n` is the number of input interfaces, and `w` is the channel width of each input interface. All input interfaces must have the same channel width. These bits are appended to either the most or least significant bits of the output channel signal.

The scheduler processes one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed.
- The input interface has no more data to send and the `valid` signal is deasserted on a ready cycle.
- When packets are supported, `endofpacket` is asserted.

Multiplexer Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Multiplexer Output Interface

The output interface carries the multiplexed data stream with data from the inputs. The symbol, data, and error widths are the same as the input interfaces.

The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the origin of the data.

You can configure the following parameters for the output interface:

- **Data Bits Per Symbol**—The bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**— The number of bits Qsys uses for the channel signal for output interfaces. For example, set this parameter to 1 if you have two input interfaces with no channel, or set this parameter to 2 if you have two input interfaces with a channel width of 1 bit. The input channel can have a width between 0-31 bits.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of `0` means the `error` signal is not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Multiplexer Parameters

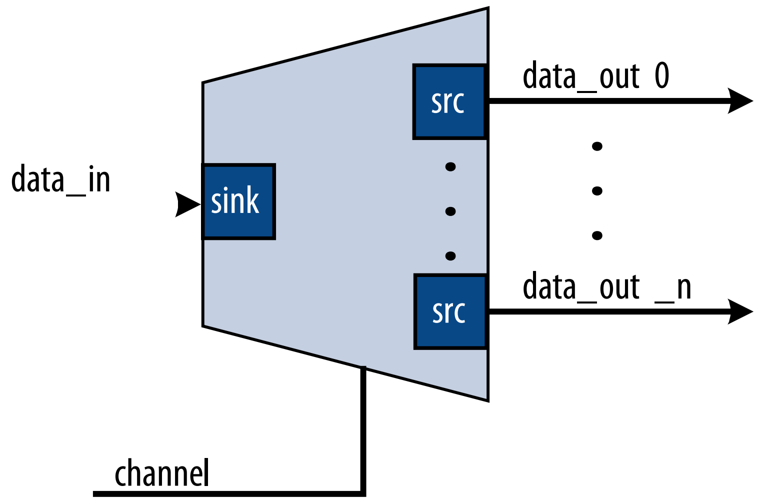
You can configure the following parameters for the multiplexer:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2 to 16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.
- **Use Packet Scheduling**—When this parameter is turned on, the multiplexer only switches the selected input interface on packet boundaries. Therefore, packets on the output interface are not interleaved.
- **Use high bits to indicate source port**—When this parameter is turned on, the multiplexer uses the high bits of the output `channel` signal to indicate the origin of the input interface of the data. For example, if the input interfaces have 4-bit channel signals, and the multiplexer has 4 input interfaces, the output interface has a 6-bit channel signal. If this parameter is turned on, bits [5:4] of the output channel signal indicate origin of the input interface of the data, and bits [3:0] are the channel bits that were presented at the input interface.

Avalon-ST Demultiplexer

Figure 10-32: Avalon-ST Demultiplexer

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal.



The data is delivered to the output interfaces in the same order it is received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface; each output interface is idle when the demultiplexer is driving data to a different output interface. The demultiplexer uses $\log_2$ (`num_output_interfaces`) bits of the `channel` signal to select the output for the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Demultiplexer Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets. You can configure the following parameters for the input interface:

- **Data Bits Per Symbol**—The bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits for the `channel` signal for output interfaces. A value of `0` means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of `0` means the `error` signal is in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Demultiplexer Output Interface

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that the demultiplexer uses to select the output interface.

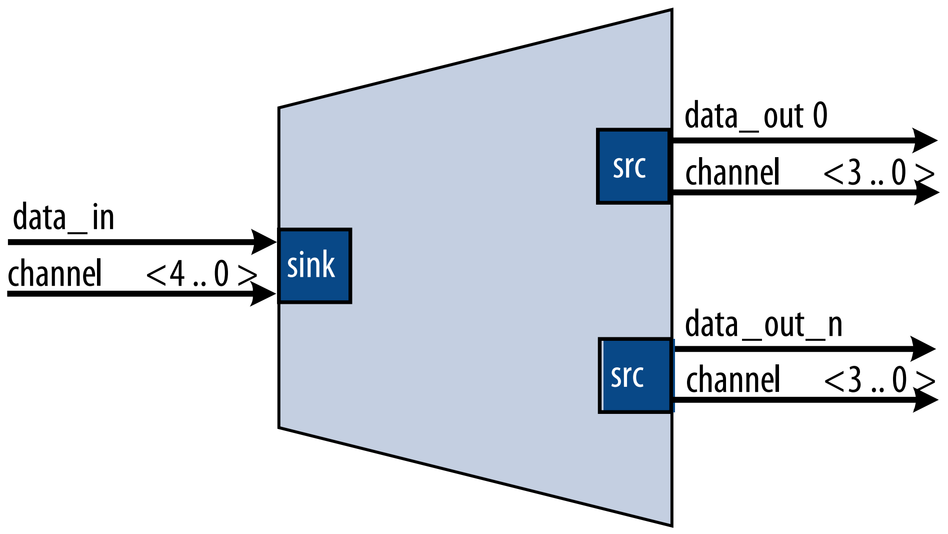
Demultiplexer Parameters

You can configure the following parameters for the demultiplexer:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2 to 16.
- **High channel bits select output**—When this option is turned on, the demultiplexing function uses the high bits of the input `channel` signal, and the low order bits are passed to the output. When this option is turned off, the demultiplexing function uses the low order bits, and the high order bits are passed to the output.

Where you place the signals in our design affects the functionality; for example, there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0, and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0 to 7 goes to channel 0 and channels 8 to 15 goes to channel 1.

Figure 10-33: Select Bits for the Demultiplexer



Single-Clock and Dual-Clock FIFO Cores

The Avalon-ST Single-Clock and Avalon-ST Dual-Clock FIFO cores are FIFO buffers which operate with a common clock and independent clocks for input and output ports respectively.

Figure 10-34: Avalon-ST Single Clock FIFO Core

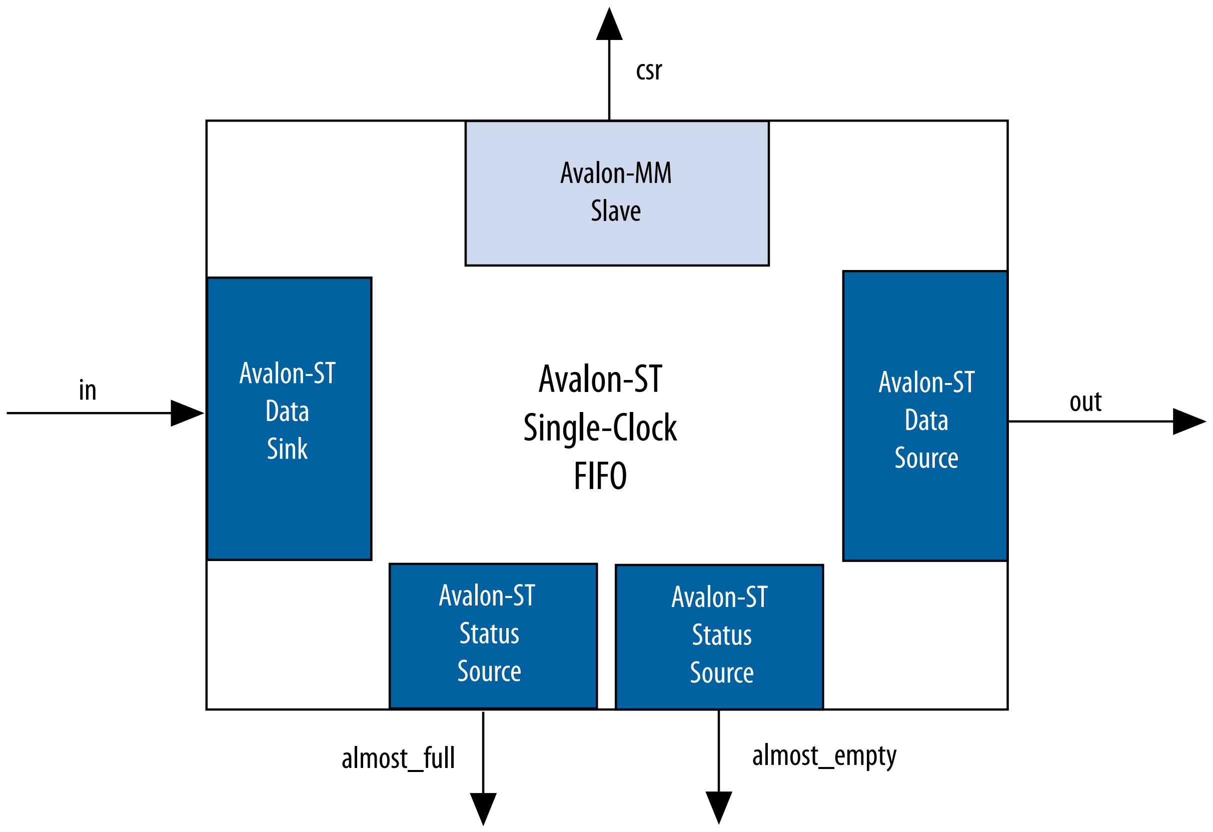
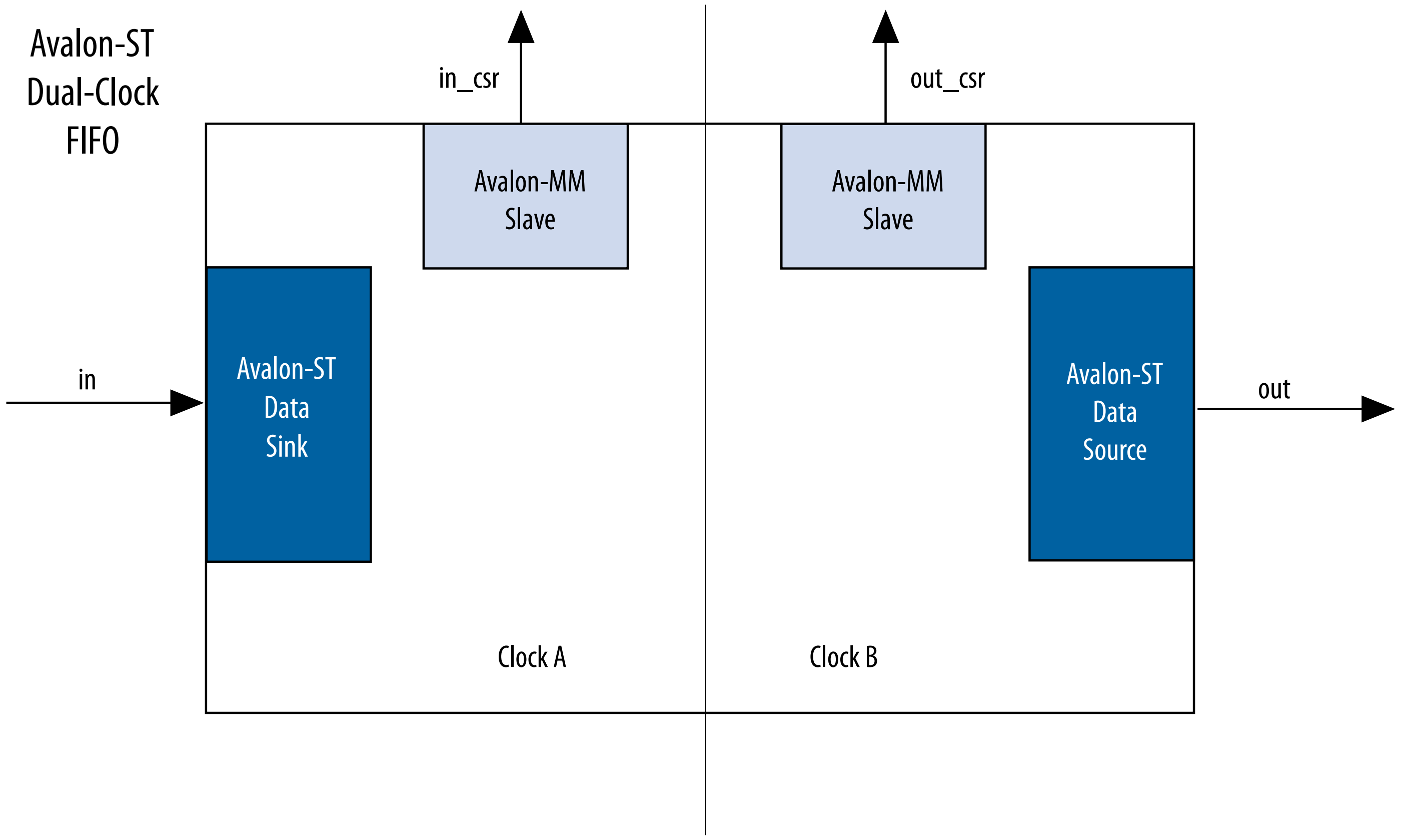


Figure 10-35: Avalon-ST Dual Clock FIFO Core



Interfaces Implemented in FIFO Cores

The following interfaces are implemented in FIFO cores:

Avalon-ST Data Interface on page 10-79

Avalon-MM Control and Status Register Interface on page 10-80

Avalon-ST Status Interface on page 10-80

Avalon-ST Data Interface

Each FIFO core has an Avalon-ST data sink and source interfaces. The data sink and source interfaces in the dual-clock FIFO core are driven by different clocks.

Table 10-69: Avalon-ST Interfaces Properties

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.

Feature	Property
Channel	Supported, up to 255 channels.
Error	Configurable.
Packet	Configurable.

Avalon-MM Control and Status Register Interface

You can configure the single-clock FIFO core to include an optional Avalon-MM interface, and the dual-clock FIFO core to include an Avalon-MM interface in each clock domain. The Avalon-MM interface provides access to 32-bit registers, which allows you to retrieve the FIFO buffer fill level and configure the almost-empty and almost-full thresholds. In the single-clock FIFO core, you can also configure the packet and error handling modes.

Avalon-ST Status Interface

The single-clock FIFO core has two optional Avalon-ST status source interfaces from which you can obtain the FIFO buffer almost-full and almost empty statuses.

FIFO Operating Modes

- **Default mode**—The core accepts incoming data on the `in` interface (Avalon-ST data sink) and forwards it to the `out` interface (Avalon-ST data source). The core asserts the `valid` signal on the Avalon-ST source interface to indicate that data is available at the interface.
- **Store and forward mode**—This mode applies only to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface only when a full packet of data is available at the interface. In this mode, you can also enable the drop-on-error feature by setting the `drop_on_error` register to 1. When this feature is enabled, the core drops all packets received with the `in_error` signal asserted.
- **Cut-through mode**—This mode applies only to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface to indicate that data is available for consumption when the number of entries specified in the `cut_through_threshold` register are available in the FIFO buffer.

Note: To turn on **Cut-through mode**, **Use store and forward** must be set to 0. Turning on **Use store and forward mode** prompts the user to turn on **Use fill level**, and then the CSR appears.

Fill Level of the FIFO Buffer

You can obtain the fill level of the FIFO buffer via the optional Avalon-MM control and status interface. Turn on the **Use fill level** parameter (**Use sink fill level** and **Use source fill level** in the dual-clock FIFO core) and read the `fill_level` register.

The dual-clock FIFO core has two fill levels. one in each clock domain. Due to the latency of the clock crossing logic, the fill levels reported in the input and output clock domains may be different for any instance. In both cases, the fill level may report badly for the clock domain; that is, the fill level is reported high in the input clock domain, and low in the output clock domain.

The dual-clock FIFO has an output pipeline stage to improve f_{MAX}. This output stage is accounted for when calculating the output fill level, but not when calculating the input fill level. Therefore, the best measure of the amount of data in the FIFO is by the fill level in the output clock domain. The fill level in the input clock domain represents the amount of space available in the FIFO (available space = FIFO depth – input fill level).

Almost-Full and Almost-Empty Thresholds to Prevent Overflow and Underflow

You can use almost-full and almost-empty thresholds as a mechanism to prevent FIFO overflow and underflow. This feature is available only in the single-clock FIFO core. To use the thresholds, turn on the **Use fill level**, **Use almost-full status**, and **Use almost-empty status** parameters. You can access the `almost_full_threshold` and `almost_full_threshold` registers via the csr interface and set the registers to an optimal value for your application.

You can obtain the almost-full and almost-empty statuses from `almost_full` and `almost_empty` interfaces (Avalon-ST status source). The core asserts the `almost_full` signal when the fill level is equal to or higher than the almost-full threshold. Likewise, the core asserts the `almost_empty` signal when the fill level is equal to or lower than the almost-empty threshold.

Single-Clock and Dual-Clock FIFO Core Parameters

Table 10-70: Single-Clock and Dual-Clock FIFO Core Parameters

Parameter	Legal Values	Description
Bits per symbol	1–32	These parameters determine the width of the FIFO.
Symbols per beat	1–32	FIFO width = **Bits per symbol** * **Symbols per beat**, where: **Bits per symbol** is the number of bits in a symbol, and **Symbols per beat** is the number of symbols transferred in a beat.
Error width	0–32	The width of the `error` signal.
FIFO depth	2^n	The FIFO depth. An output pipeline stage is added to the FIFO to increase performance, which increases the FIFO depth by one. <n> = n=1,2,3,4...
Use packets	—	Turn on this parameter to enable data packet support on the Avalon-ST data interfaces.
Channel width	1–32	The width of the `channel` signal.
Avalon-ST Single Clock FIFO Only		
Use fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface (CSR). The CSR is enabled when **Use fill level** is set to 1.

Parameter	Legal Values	Description
Use Store and Forward		To turn on **Cut-through mode**, **Use store and forward** must be set to 0. Turning on **Use store and forward** prompts the user to turn on **Use fill level**, and then the CSR appears.
Avalon-ST Dual Clock FIFO Only		
Use sink fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.
Use source fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.
Write pointer synchronizer length	2–8	The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.
Use Max Channel	—	Turn on this parameter to specify the maximum channel number.
Max Channel	1–255	Maximum channel number.

Note: For more information on metastability in Altera devices, refer to *Understanding Metastability in FPGAs*. For more information on metastability analysis and synchronization register chains, refer to the *Managing Metastability*.

Related Information

Understanding Metastability in FPGAs

Managing Metastability on page 13-1

Avalon-ST Single-Clock FIFO Registers

Table 10-71: Avalon-ST Single-Clock FIFO Registers

The CSR interface in the Avalon-ST Single Clock FIFO core provides access to registers.

32-Bit Word Offset	Name	Access	Reset	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are not used.
1	Reserved	—	—	Reserved for future use.
2	`almost_full_threshold`	RW	**FIFO depth**–1	Set this register to a value that indicates the FIFO buffer is getting full.
3	`almost_empty_threshold`	RW	0	Set this register to a value that indicates the FIFO buffer is getting empty.
4	`cut_through_threshold`	RW	0	**0**—Enables store and forward mode. **Greater than 0**—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the `valid` signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet. **Note:** To turn on **Cut-through mode**, **Use store and forward** must be set to 0. Turning on **Use store and forward mode** prompts the user to turn on **Use fill level**, and then the CSR appears.
5	`drop_on_error`	RW	0	**0**—Disables drop-on error. **1**—Enables drop-on error. This register applies only when the **Use packet** and **Use store and forward** parameters are turned on.

Table 10-72: Register Description for Avalon-ST Dual-Clock FIFO

The `in_csr` and `out_csr` interfaces in the Avalon-ST Dual Clock FIFO core reports the FIFO fill level.

32-Bit Word Offset	Name	Access	Reset Value	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are not used.

Related Information

- Avalon Interface Specifications
- Avalon Memory-Mapped Design Optimizations

Document Revision History

Table 10-73: Document Revision History

The table below indicates edits made to the *Qsys System Design Components* content since its creation.

Date	Version	Changes
2015.05.04	15.0.0	Avalon-MM Unaligned Burst Expansion Bridge and Avalon-MM Pipeline Bridge, **Maximum pending read transactions** parameter. Extended description.
December 2014	14.1.0	• AXI Timout Bridge. • Added notes to *Avalon-MM Clock Crossing Bridge* pertaining to: • SDC constraints for its internal asynchronous FIFOs. • FIFO-based clock crossing.
June 2014	14.0.0	• AXI Bridge support. • Address Span Extender updates. • Avalon-MM Unaligned Burst Expansion Bridge support.
November 2013	13.1.0	• Address Span Extender
May 2013	13.0.0	• Added Streaming Pipeline Stage support. • Added AMBA APB support.
November 2012	12.1.0	• Moved relevant content from the *Embedded Peripherals IP User Guide*.

Related Information

Quartus II Handbook Archive

Recommended Design Practices 11

2014.12.15

QII5V1 Subscribe Send Feedback

This chapter provides design recommendations for Altera® devices and describes the Quartus® II Design Assistant, which helps you check your design for violations of Altera's design recommendations.

Current FPGA applications have reached the complexity and performance requirements of ASICs. In the development of complex system designs, good design practices have an enormous impact on the timing performance, logic utilization, and system reliability of a device. Well-coded designs behave in a predictable and reliable manner even when retargeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and ASIC implementations for prototyping and production.

For optimal performance, reliability, and faster time-to-market when designing with Altera devices, you should adhere to the following guidelines:

- Understand the impact of synchronous design practices
- Follow recommended design techniques, including hierarchical design partitioning, and timing closure guidelines
- Take advantage of the architectural features in the targeted device

Following Synchronous FPGA Design Practices

The first step in good design methodology is to understand the implications of your design practices and techniques. This section outlines the benefits of optimal synchronous design practices and the hazards involved in other techniques.

Good synchronous design practices can help you meet your design goals consistently. Problems with other design techniques can include reliance on propagation delays in a device, which can lead to race conditions, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers every event. As long as you ensure that all the timing requirements of the registers are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily migrate synchronous designs to different device families or speed grades.

Implementing Synchronous Designs

In a synchronous design, the clock signal controls the activities of all inputs and outputs.

On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, the outputs of combinational logic feeding the data inputs of registers change values. This change triggers a period of instability due to propagation delays through the logic as the signals go through several transitions and finally settle to new values. Changes that occur on data inputs of registers do not affect the values of their outputs until after the next active clock edge.

Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as you meet the following timing requirements:

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

  When you specify all of your clock frequencies and other timing requirements, the Quartus II TimeQuest Timing Analyzer reports actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin in your design. By meeting these external pin requirements and following synchronous design techniques, you ensure that you satisfy the setup and hold times for all registers in your device.

  Tip: To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feed a register should have a synchronous relationship with the clock of the register. If signals are asynchronous, you can register the signals at the inputs of the device to help prevent a violation of the required setup and hold times.

  When you violate the setup or hold time of a register, you might oscillate the output, or set the output to an intermediate voltage level between the high and low levels called a metastable state. In this unstable state, small perturbations such as noise in power rails can cause the register to assume either the high or low voltage level, resulting in an unpredictable valid state. Various undesirable effects can occur, including increased propagation delays and incorrect output states. In some cases, the output can even oscillate between the two valid states for a relatively long period of time.

Related Information

About TimeQuest Timing Analysis

For information about timing requirements and analysis in the Quartus II software, refer to About TimeQuest Timing Analysis in Quartus II Help.

Asynchronous Design Hazards

Designers use asynchronous techniques such as ripple counters or pulse generators in programmable logic device (PLD) designs, enabling them to take “short cuts” to save device resources.

Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which can vary with temperature and voltage fluctuations, resulting in incomplete timing constraints and possible glitches and spikes.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. In these cases, race conditions can arise where the order of signal changes can affect the output of the logic. PLD designs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine the timing delay associated with a particular block of logic ahead of time. As devices become faster due to device process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. This chapter provides specific examples. Relying on a particular delay also makes asynchronous designs difficult to migrate to different architectures, devices, or speed grades.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, the timing-driven algorithms used by your synthesis and place-and-route tools may not be able to perform the best optimizations, and the reported results may not be complete.

Some asynchronous design structures can generate harmful glitches, which are pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit several glitches before they settle to their new values. These glitches can propagate through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

HDL Design Guidelines

When designing with HDL code, you should understand how a synthesis tool interprets different HDL design techniques and what results to expect.

Your design techniques can affect logic utilization and timing performance, as well as the design's reliability. This section describes basic design techniques that ensure optimal synthesis results for designs targeted to Altera devices while avoiding several common causes of unreliability and instability. Altera recommends that you design your combinational logic carefully to avoid potential problems and pay attention to your clocking schemes so that you can maintain synchronous functionality and avoid timing problems.

Optimizing Combinational Logic

Combinational logic structures consist of logic functions that depend only on the current state of the inputs. In Altera FPGAs, these functions are implemented in the look-up tables (LUTs) with either logic elements (LEs) or adaptive logic modules (ALMs).

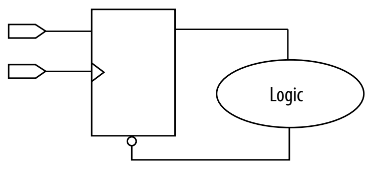
For cases where combinational logic feeds registers, the register control signals can implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

Avoid Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs. Combinational loops generally violate synchronous design principles by establishing a direct feedback loop that contains no registers.

You should avoid combinational loops whenever possible. In a synchronous design, feedback loops should include registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side in HDL code. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic.

Figure 11-1: Combinational Loop Through Asynchronous Control Pin



Tip: Use recovery and removal analysis to perform timing analysis on asynchronous ports, such as `clear` or `reset` in the Quartus II software.

Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change, which means the behavior of the loop is unpredictable.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinational loops to process the design. The various tools used in the design flow may open a given loop in a different manner, processing it in a way that is inconsistent with the original design intent.

Related Information

Specifying Timing Constraints and Exceptions

Avoid Unintended Latch Inference

A latch is a small circuit with combinational feedback that holds a value until a new value is assigned. You can implement latches with the Quartus II Text Editor or Block Editor.

It is common for mistakes in HDL code to cause unintended latch inference; Quartus II Synthesis issues a warning message if this occurs. Unlike other technologies, a latch in FPGA architecture is not significantly smaller than a register. The architecture is not optimized for latch implementation and latches generally have slower timing performance compared to equivalent registered circuitry.

Latches have a transparent mode in which data flows continuously from input to output. A positive latch is in transparent mode when the enable signal is high (low for negative latch). In transparent mode, glitches on the input can pass through to the output because of the direct path created. This presents significant complexity for timing analysis. Typical latch schemes use multiple enable phases to prevent long transparent paths from occurring. However, timing analysis cannot identify these safe applications.

The TimeQuest analyzer analyzes latches as synchronous elements clocked on the falling edge of the positive latch signal by default, and allows you to treat latches as having nontransparent start and end points. Be aware that even an instantaneous transition through transparent mode can lead to glitch propagation. The TimeQuest analyzer cannot perform cycle-borrowing analysis.

Due to various timing complexities, latches have limited support in formal verification tools. Therefore, you should not rely on formal verification for a design that includes latches.

Tip: Avoid using latches to ensure that you can completely analyze the timing performance and reliability of your design.

Avoid Delay Chains in Clock Paths

You require delay chains when you use two or more consecutive nodes with a single fan-in and a single fan-out to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

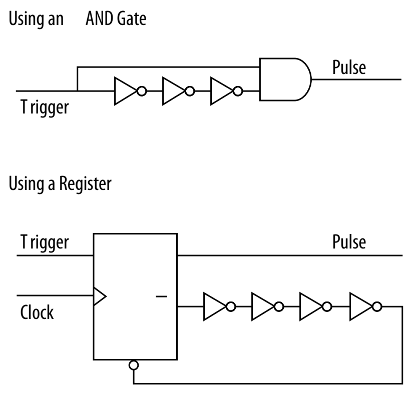
Delays in PLD designs can change with each placement and routing cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Avoid using delay chains to prevent these kinds of problems.

In some ASIC designs, delays are used for buffering signals as they are routed around the device. This functionality is not required in FPGA devices because the routing structure provides buffers throughout the device.

Use Synchronous Pulse Generators

You can use delay chains to generate either one pulse (pulse generators) or a series of pulses (multivibrators). There are two common methods for pulse generation. These techniques are purely asynchronous and must be avoided.

Figure 11-2: Asynchronous Pulse Generators



A trigger signal feeds both inputs of a 2-input AND gate, but the design adds inverts to create a delay chain to one of the inputs. The width of the pulse depends on the time differences between path that feeds the gate directly, and the path that goes through the delay chain. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of input values. This technique artificially increases the width of the glitch.

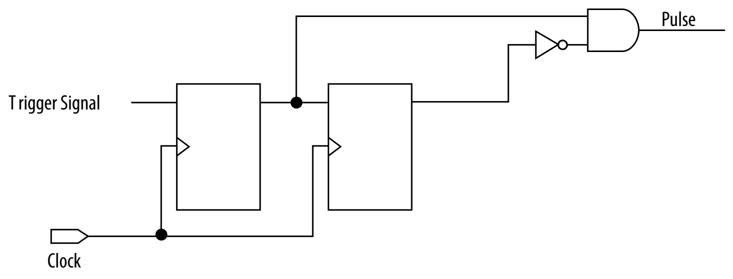
A register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

The width of pulses generated in this way are difficult for synthesis and place-and-route to determine, set, or verify. The actual pulse width can only be determined after placement and routing, when routing and propagation delays are known. You cannot reliably create a specific pulse width when creating HDL code, and it cannot be set by EDA tools. The pulse may not be wide enough for the application under all PVT conditions. Also, the pulse width changes if you change to a different device. Additionally, verification is difficult because static timing analysis cannot verify the pulse width.

Multivibrators use a glitch generator to create pulses, together with a combinational loop that turns the circuit into an oscillator. This creates additional problems because of the number of pulses involved. Additionally, when the structures generate multiple pulses, they also create a new artificial clock in the design must be analyzed by design tools.

When you must use a pulse generator, use synchronous techniques.

Figure 11-3: Recommended Pulse-Generation Technique



The pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily moved to other architectures, devices, or speed grades.

Optimizing Clocking Schemes

Like combinational logic, clocking schemes have a large effect on the performance and reliability of a design.

Avoid using internally generated clocks (other than PLLs) wherever possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that create functional problems, and the delay inherent in combinational logic can lead to timing problems.

Tip: Specify all clock relationships in the Quartus II software to allow for the best timing-driven optimizations during fitting and to allow correct timing analysis. Use clock setting assignments on any derived or internal clocks to specify their relationship to the base clock.

Use global device-wide, low-skew dedicated routing for all internally-generated clocks, instead of routing clocks on regular routing lines.

Avoid data transfers between different clocks wherever possible. If you require a data transfer between different clocks, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a clock setup uncertainty and clock hold uncertainty value of 10% to 15% of the clock delay.

The following sections provide specific examples and recommendations for avoiding clocking scheme problems.

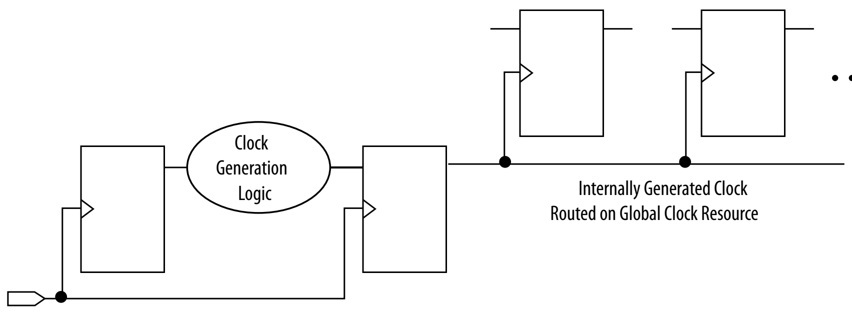
Register Combinational Logic Outputs

If you use the output from combinational logic as a clock signal or as an asynchronous reset signal, you can expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or an asynchronous input) to a register can have significant consequences.

Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold requirements might also be violated if the data input of the register changes when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

To avoid these problems, you should always register the output of combinational logic before you use it as a clock signal.

Figure 11-4: Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that glitches generated by the combinational logic are blocked at the data input of the register.

Avoid Asyncrhonous Clock Division

Designs often require clocks that you create by dividing a master clock. Most Altera FPGAs provide dedicated phase-locked loop (PLL) circuitry for clock division. Using dedicated PLL circuitry can help you to avoid many of the problems that can be introduced by asynchronous clock division logic.

When you must use logic to divide a master clock, always use synchronous counters or state machines. Additionally, create your design so that registers always directly generate divided clock signals, and route the clock on global clock resources. To avoid glitches, do not decode the outputs of a counter or a state machine to generate clock signals.

Avoid Ripple Counters

To simplify verification, avoid ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts.

Ripple counters use cascaded registers, in which the output pin of one register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks must be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and placement and routing tools.

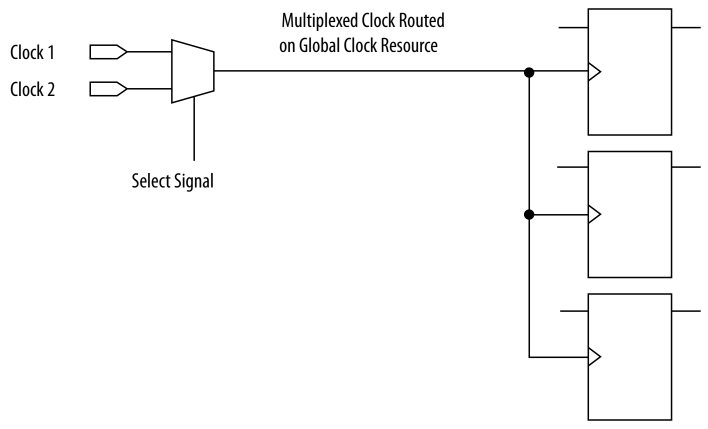
You can often use ripple clock structures to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus II software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows you to construct a counter using one logic element per counter bit. You should avoid using ripple counters completely.

Use Multiplexed Clocks

Use clock multiplexing to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source.

For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 11-5: Multiplexing Logic and Clock Sources



Adding multiplexing logic to the clock signal can create the problems addressed in the previous sections, but requirements for multiplexed clocks vary widely, depending on the application. Clock multiplexing is acceptable when the clock signal uses global clock routing resources and if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- Registers are always reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks in real time with no reset signal, and your design cannot tolerate a temporarily incorrect response, you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems. By default, the Quartus II software optimizes and analyzes all possible paths through the multiplexer and between both internal clocks that may come from the multiplexer. This may lead to more restrictive analysis than required if the multiplexer is always selecting one particular clock. If you do not require the more complete analysis, you can assign the output of the multiplexer as a base clock in the Quartus II software, so that all register-to-register paths are analyzed using that clock.

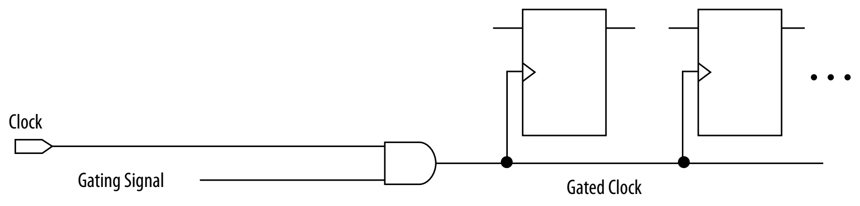
Tip: Use dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the clock-switchover feature or clock control block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing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

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

Use Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls gating circuitry. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 11-6: Gated Clock



You can use gated clocks to reduce power consumption in some device architectures by effectively shutting down portions of a digital circuit when they are not in use. When a clock is gated, both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

Use dedicated hardware to perform clock gating rather than an AND or OR gate. For example, you can use the clock control block in newer Altera devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew, and avoid any possible hold time problems on the device due to logic delay on the clock line.

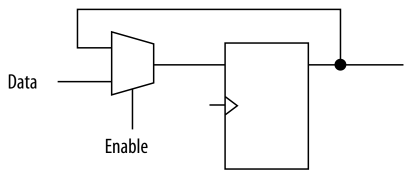
From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, use a synchronous scheme.

Use Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use a synchronous clock enable signal. FPGAs efficiently support clock enable signals because there is a dedicated clock enable signal available on all device registers.

This scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, and performs the same function as a gated clock by disabling a set of registers. Insert a multiplexer in front of the data input of every register to either load new data, or copy the output of the register.

Figure 11-7: Synchronous Clock Enable



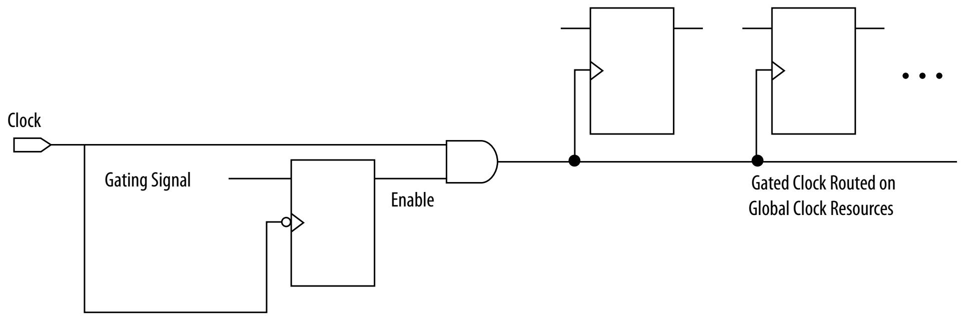
Recommended Clock-Gating Methods

Use gated clocks only when your target application requires power reduction and when gated clocks are able to provide the required reduction in your device architecture.

If you must use clocks gated by logic, implement these clocks using the robust clock-gating technique and ensure that the gated clock signal uses dedicated global clock routing.

You can gate a clock signal at the source of the clock network, at each register, or somewhere in between. Because the clock network contributes to switching power consumption, gate the clock at the source whenever possible, so that you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 11-8: Recommended Clock-Gating Technique



A register generates the enable signal to ensure that the signal is free of glitches and spikes. The register that generates the enable signal is triggered on the inactive edge of the clock to be gated. Use the falling edge when gating a clock that is active on the rising edge. Using this technique, only one input of the gate that turns the clock on and off changes at a time. This prevents glitches or spikes on the output. Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable command with a positive edge-triggered register.

When using this technique, pay close attention to the duty cycle of the clock and the delay through the logic that generates the enable signal because you must generate the enable command in one-half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, careful management of the duty cycle and logic delay may be an acceptable solution when compared with problems created by other methods of gating clocks.

Ensure that you apply a clock setting to the gated clock in the TimeQuest analyzer. Apply a clock setting to the output of the AND gate. Otherwise, the timing analyzer might analyze the circuit using the clock path through the register as the longest clock path and the path that skips the register as the shortest clock path, resulting in artificial clock skew.

In certain cases, converting the gated clocks to clock enables may help reduce glitch and clock skew, and eventually produce a more accurate timing analysis. You can set the Quartus II software to automatically convert gated clocks to clock enables by turning on the **Auto Gated Clock Conversion** option. The conversion applies to two types of gated clocking schemes: single-gated clock and cascaded-gated clock.

Optimizing Physical Implementation and Timing Closure

This section provides design and timing closure techniques for high speed or complex core logic designs with challenging timing requirements. These techniques may also be helpful for low or medium speed designs.

Planning Physical Implementation

When planning a design, consider the following elements of physical implementation:

- The number of unique clock domains and their relationships
- The amount of logic in each functional block
- The location and direction of data flow between blocks
- How data routes to the functional blocks between I/O interfaces

  Interface-wide control or status signals may have competing or opposing constraints. For example, when a functional block's control or status signals interface with physical channels from both sides of the device. In such cases you must provide enough pipeline register stages to allow these signals to traverse the width of the device. In addition, you can structure the hierarchy of the design into separate logic modules for each side of the device. The side modules can generate and use registered control signals per side. This simplifies floorplanning, particularly in designs with transceivers, by placing per-side logic near the transceivers.

  When adding register stages to pipeline control signals, turn off the **Auto Shift Register Replacement** option (**Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**) for these registers. By default, chains of registers can be converted to a RAM-based implementation based on performance and resource estimates. Since pipelining helps meet timing requirements over long distance, this assignment ensures that control signals are not converted.

Planning FPGA Resources

Your design requirements impact the use of FPGA resources. Plan functional blocks with appropriate global, regional, and dual-regional network signals in mind.

In general, after allocating the clocks in a design, use global networks for the highest fan-out control signals. When a global network signal distributes a high fan-out control signal, the global signal can drive logic anywhere in the device. Similarly, when using a regional network signal, the driven must be in one quadrant of the device, or half the device for a dual-regional network signal. Depending on data flow and physical locations of the data entry and exit between the I/Os and the device, restricting a functional block to a quadrant or half the device may not be practical for performance or resource requirements.

When floorplanning a design, consider the balance of different types of device resources, such as memory, logic, and DSP blocks in the main functional blocks. For example, if a design is memory intensive with a small amount of logic, it may be difficult to develop an effective floorplan. Logic that interfaces with the memory would have to spread across the chip to access the memory. In this case, it is important to use enough register stages in the data and control paths to allow signals to traverse the chip to access the physically disparate resources needed.

Optimizing Timing Closure

You can make changes to your design and constraints that help you achieve timing closure.

Whenever you change the project settings, you must balance any performance improvement of the setting against any potential increase in compilation time associated with the setting. You can view the performance gain versus runtime cost by reviewing the Fitter messages after design processing.

You can use physical synthesis optimizations for combinational logic, register retiming, and register duplication techniques to optimize your design for timing closure.

Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)** to turn on physical synthesis options.

- Physical synthesis for combinational logic—When the **Perform physical synthesis for combinational logic** is turned on, the report panel identifies logic that physical synthesis can modify. You can use this information to modify the design so that the associated optimization can be turned off to save compile time.
- Register duplication—This technique is most useful where registers have high fan-out, or where the fan-out is in physically distant areas of the device. Review the netlist optimizations report and consider manually duplicating registers automatically added by physical synthesis. You can also locate the original and duplicate registers in the Chip Planner. Compare their locations, and if the fan-out is improved, modify the code and turn off register duplication to save compile time.
- Register retiming—This technique is particularly useful where some combinatorial paths between registers exceed the timing goal while other paths fall short. If a design is already heavily pipelined, register retiming is less likely to provide significant performance gains since there should not be significantly unbalanced levels of logic across pipeline stages.

  The application of appropriate timing constraints is essential to timing closure. Use the following general guidelines in applying timing constraints:
- Apply multicycle constraints in your design wherever single-cycle timing analysis is not required.
- Apply False Path constraints to all asynchronous clock domain crossings or resets in the design. This technique prevents overconstraining and the Fitter focuses only on critical paths to reduce compile time. However, over constraining timing critical clock domains can sometimes provide better timing results and lower compile times than physical synthesis.
- Overconstrain rather than using physical synthesis when the slack improvement from physical synthesis is near zero. Overconstrain the frequency requirement on timing critical clock domains by using setup uncertainty.
- When evaluating the effect of constraint changes on performance and runtime, compile the design with at least three different seeds to determine the average performance and runtime effects. Different constraint combinations produce various results. Three samples or more establishes a performance trend. Modify your constraints based on performance improvement or decline.
- Leave settings at the default value whenever possible. Increasing performance constraints can increase the compile time significantly. While those increases may be necessary to close timing on a design, using the default settings whenever possible minimizes compile time.

Optimizing Critical Timing Paths

To close timing in high speed designs, review paths with the largest timing failures. Correcting a single, large timing failure can result in a very significant timing improvement.

Review the register placement and routing paths by clicking **Tools** > **Chip Planner**. Large timing failures on high fan-out control signals can be caused by any of the following conditions:

- Sub-optimal use of global networks
- Signals that traverse the chip on local routing without pipelining
- Failure to correct high fan-out by register duplication

  For high-speed and high-bandwidth designs, optimize speed by reducing bus width and wire usage. To reduce wire use, move the data as little as possible. For example, if a block of logic functions on a few bits of a word, store inactive bits in a fifo or memory. Memory is cheaper and denser than registers and reduces wire usage.

Optimizing Power Consumption

The total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. Knowledge of the relationship between these components is fundamental in calculating the overall total power consumption.

You can use various optimization techniques and tools to minimize power consumption when applied during FPGA design implementation. The Quartus II software offers power-driven compilation features to fully optimize device power consumption. Power-driven compilation focuses on reducing your design's total power consumption using power-driven synthesis and power-driven placement and routing.

Managing Design Metastability

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

Related Information

Viewing Metastability Reports

For more information about viewing metastability reports, refer to Viewing Metastability Reports in Quartus II Help.

Checking Design Violations

To improve the reliability, timing performance, and logic utilization of your design, avoid design rule violations. The Quartus II software provides the Design Assistant tool that automatically checks for design rule violations and reports their location.

The Design Assistant is a design rule checking tool that allows you to check for design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. You can specify which rules you want the Design Assistant to apply to your design. This is useful if you know that your design violates particular rules that are not critical and you can allow these rule violations. The Design Assistant generates design violation reports with details about each violation based on the settings that you specified.

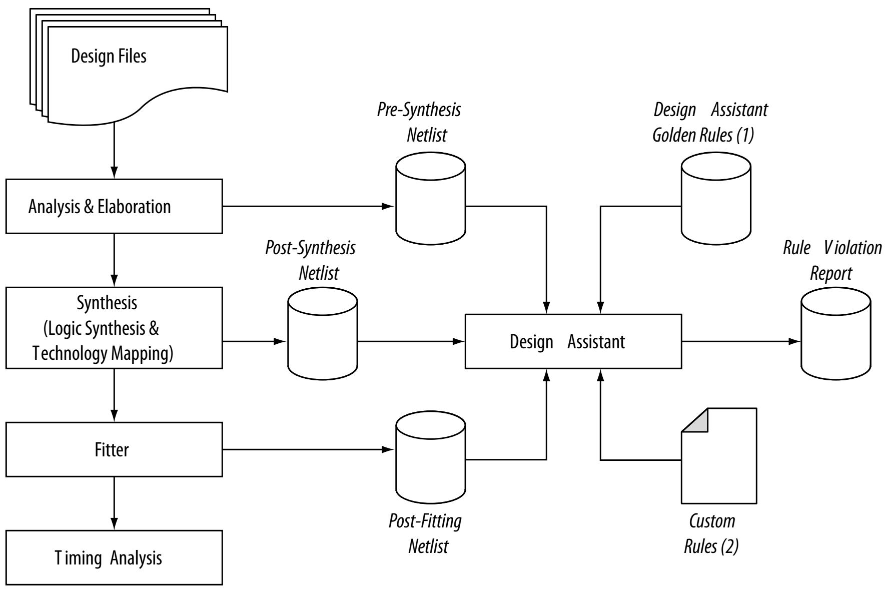
This section provides an introduction to the Quartus II design flow with the Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last parts of the section describe the design rules and the reports generated by the Design Assistant. The Design Assistant supports all Altera devices supported by the Quartus II software.

Validating Against Design Rules

You can run the Design Assistant following design synthesis or compilation. The Design Assistant performs a post-fit netlist analysis of your design.

The default is to apply all of the rules to your project. If there are some rules that are unimportant to your design, you can turn off the rules that you do not want the Design Assistant to use.

Figure 11-9: Quartus II Design Flow with the Design Assistant



1. Database of the default rules for the Design Assistant.
2. A file that contains the **.xml** codes of the custom rules for the Design Assistant. For more details about how to create this file .

   The Design Assistant analyzes your design netlist at different stages of the compilation flow and may yield different warnings or errors, even though the netlists are functionally the same. Your pre-synthesis, post-synthesis, and post-fitting netlists might be different due to optimizations performed by the Quartus II software. For example, a warning message in a pre-synthesis netlist may be removed after the netlist has been synthesized into a post-synthesis or post-fitting netlist.

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

- When you run the Design Assistant after running a full compilation or fitting, the Design Assistant performs a post-fitting analysis on the design.
- When you run the Design Assistant after performing Analysis and Synthesis, the Design Assistant performs post-synthesis analysis on the design.
- When you start the Design Assistant after performing Analysis and Elaboration, the Design Assistant performs a pre-synthesis analysis on the design. You can also perform pre-synthesis analysis with the Design Assistant using the command-line. You can use the `-rtl` option with the `quartus_drc` executable, as shown in the following example:

  ```
  quartus_drc <project_name> --rtl=on
  ```

  If your design violates a design rule, the Design Assistant generates warning messages and information messages about the violated rule. The Design Assistant displays these messages in the Messages window, in the Design Assistant Messages report, and in the Design Assistant report files. You can find the Design Assistant report files called <project_name>**.drc.rpt** in the <project_name> subdirectory of the project directory.

Related Information

About the Design Assistant

Creating Custom Design Rules

You can define and validate your design against your own custom set of design rules. You can save these rules in a text file (with any file extension) with the XML format.

You then specify the path to that file in the Design Assistant settings page and run the Design Assistant for violation checking.

Refer to the following location to locate the file that contains the default rules for the Design Assistant:

<Quartus II install path>**\quartus\libraries\design-assistant\da_golden_rule.xml**

Custom Design Rule Examples

The following examples of custom rules show how to check node relationships and clock relationships in a design.

This example shows the XML codes for checking SR latch structures in a design.

Example 11-1: Detecting SR Latches in a Design

```
<DA_RULE ID="EX01" SEVERITY="CRITICAL" NAME="Checking Design for SR Latch"
DEFAULT_RUN="YES">
<RULE_DEFINITION>
   <FORBID>
   <OR>
      <NODE NAME="NODE_1" TYPE="SRLATCH" />
      <HAS_NODE NODE_LIST="NODE_1" />
      <NODE NAME="NODE_1" TOTAL_FANIN="EQ2" />
      <NODE NAME="NODE_2" TOTAL_FANIN="EQ2" />
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND"
TO_NAME="NODE_2" TO_TYPE="NAND" />
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND"
TO_NAME="NODE_1" TO_TYPE="NAND" />
      </AND>
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR"
TO_NAME="NODE_2" TO_TYPE="NOR" />
```

```
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR"
TO_NAME="NODE_1" TO_TYPE="NOR" />
      </AND>
      </OR>
   </FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
   <MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
      <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
      <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   </MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

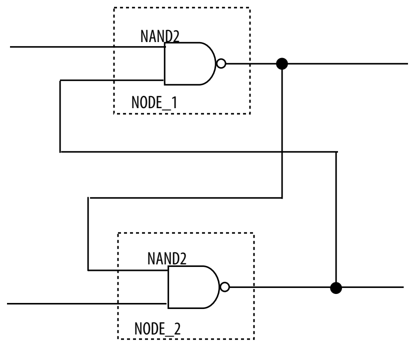
The possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the block is true, the block is fulfilled (OR gate similarity). If no `<AND></AND>` or `<OR></OR>` blocks are specified, the default is `<AND></AND>`.

The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the SR latch structures. If the condition is fulfilled, the Design Assistant highlights a rule violation.

Example 11-2: Detecting SR Latches in a Design

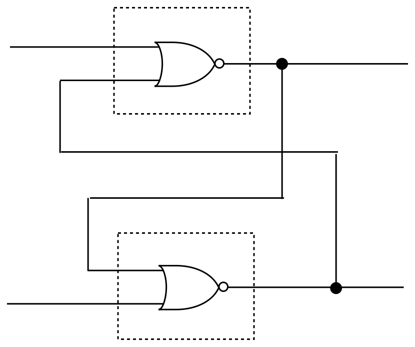
```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
   TO_TYPE="NAND" />
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
   TO_TYPE="NAND" />
</AND>
```

Figure 11-10: Undesired Condition 1



```
<AND>
 <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR" TO_NAME="NODE_2"
TO_TYPE="NOR" />
 <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR" TO_NAME="NODE_1"
TO_TYPE="NOR" />
</AND>
```

Figure 11-11: Undesired Condition 2



This example shows how to use the `CLOCK_RELATIONSHIP` attribute to relate nodes to clock domains. This example checks for correct synchronization in data transfer between asynchronous clock domains. Synchronization is done with cascaded registers, also called synchronizers, at the receiving clock domain. The code in This example checks for the synchronizer configuration based on the following guidelines:

- The cascading registers need to be triggered on the same clock edge
- There is no logic between the register output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

Example 11-3: Detecting Incorrect Synchronizer Configuration

```
<DA_RULE ID="EX02" SEVERITY="HIGH" NAME="Data Transfer Not Synch Correctly"
DEFAULT_RUN="YES">

<RULE_DEFINITION>
<DECLARE>
   <NODE NAME="NODE_1" TYPE="REG" />
   <NODE NAME="NODE_2" TYPE="REG" />
   <NODE NAME="NODE_3" TYPE="REG" />
</DECLARE>
<FORBID>
   <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />
   <NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
   <OR>
      <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2"
TO_PORT="D_PORT" REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATION-
SHIP="ASYN" />
      <CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
   </OR>
</FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
<MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
   <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
   <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   <MESSAGE NAME="Source node(s): %ARG3%, Destination node(s): %ARG4%">
      <MESSAGE_ARGUMENT NAME="ARG3" TYPE="NODE" VALUE="NODE_1" />
      <MESSAGE_ARGUMENT NAME="ARG4" TYPE="NODE" VALUE="NODE_2" />
   </MESSAGE>
```

```
</MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

The codes differentiate the clock domains. `ASYN` means asynchronous, and `!ASYN` means non-asynchronous. This notation is useful for describing nodes that are in different clock domains. The following lines from the example state that `NODE_2` and `NODE_3` are in the same clock domain, but `NODE_1` is not.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
```

The next line of code states that `NODE_2` and `NODE_3` have a clock relationship of either sequential edge or asynchronous.

```
<CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

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

The possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the block is true, the block is fulfilled (OR gate similarity). If no `<AND></AND>` or `<OR></OR>` blocks are specified, the default is `<AND></AND>`.

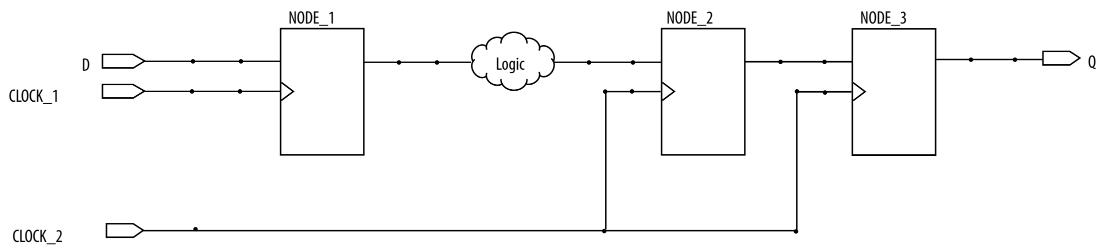
The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the SR latch structures. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples show the undesired conditions from with their equivalent block diagrams:

Example 11-4: Undesired Condition 3

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
REQUIRED_THROUGH="YES"
   THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
```

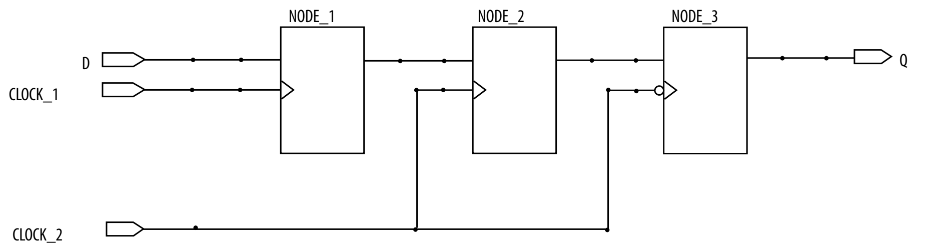
Figure 11-12: Undesired Condition 3



Example 11-5: Undesired Condition 4

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

Figure 11-13: Undesired Condition 4



Use Clock and Register-Control Architectural Features

In addition to following general design guidelines, you must code your design with the device architecture in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Use Global Clock Network Resources

Altera FPGAs provide device-wide global clock routing resources and dedicated inputs. Use the FPGA’s low-skew, high fan-out dedicated routing where available.

By assigning a clock input to one of these dedicated clock pins or with a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

In an ASIC design, you should balance the clock delay as it is distributed across the device. Because Altera FPGAs provide device-wide global clock routing resources and dedicated inputs, there is no need to manually balance delays on the clock network.

You should limit the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems. In addition, when you use combinational logic to generate an internal clock, it adds delays on the clock path. In some cases, delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, you violate the timing parameters of the register (such as hold time requirements) and the design does not function correctly.

FPGAs offer a number of low-skew global routing resources to distribute high fan-out signals to help with the implementation of large designs with many clock domains. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically several dedicated clock pins to drive either global or regional clock networks, and both PLL outputs and internal clocks can drive various clock networks.

To reduce clock skew in a given clock domain and ensure that hold times are met in that clock domain, assign each clock signal to one of the global high fan-out, low-skew clock networks in the FPGA device. The Quartus II software automatically uses global routing for high fan-out control signals, PLL outputs, and signals feeding the global clock pins on the device. You can make explicit Global Signal logic option settings by turning on the **Global Signal** option setting. Use this option when it is necessary to force the software to use the global routing for particular signals.

To take full advantage of these routing resources, the sources of clock signals in a design (input clock pins or internally-generated clocks) need to drive only the clock input ports of registers. In older Altera device families, if a clock signal feeds the data ports of a register, the signal may not be able to use dedicated routing, which can lead to decreased performance and clock skew problems. In general, allowing clock signals to drive the data ports of registers is not considered synchronous design and can complicate timing analysis.

Use Global Reset Resources

ASIC designs may use local resets to avoid long routing delays. Take advantage of the device-wide asynchronous reset pin available on most FPGAs to eliminate these problems. This reset signal provides low-skew routing across the device.

The following are three types of resets used in synchronous circuits:

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

Use Synchronous Resets

The synchronous reset ensures that the circuit is fully synchronous. You can easily time the circuit with the Quartus II TimeQuest analyzer.

Because clocks that are synchronous to each other launch and latch the reset signal, the data arrival and data required times are easily determined for proper slack analysis. The synchronous reset is easier to use with cycle-based simulators.

There are two methods by which a reset signal can reach a register; either by being gated in with the data input, or by using an LAB-wide control signal (`synclr`). If you use the first method, you risk adding an additional gate delay to the circuit to accommodate the reset signal, which causes increased data arrival times and negatively impacts setup slack. The second method relies on dedicated routing in the LAB to each register, but this is slower than an asynchronous reset to the same register.

Figure 11-14: Synchronous Reset

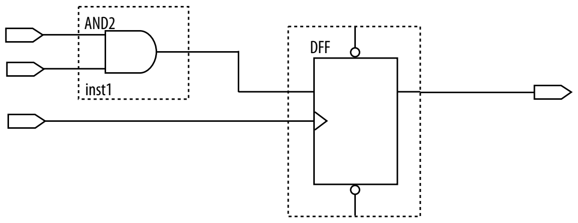
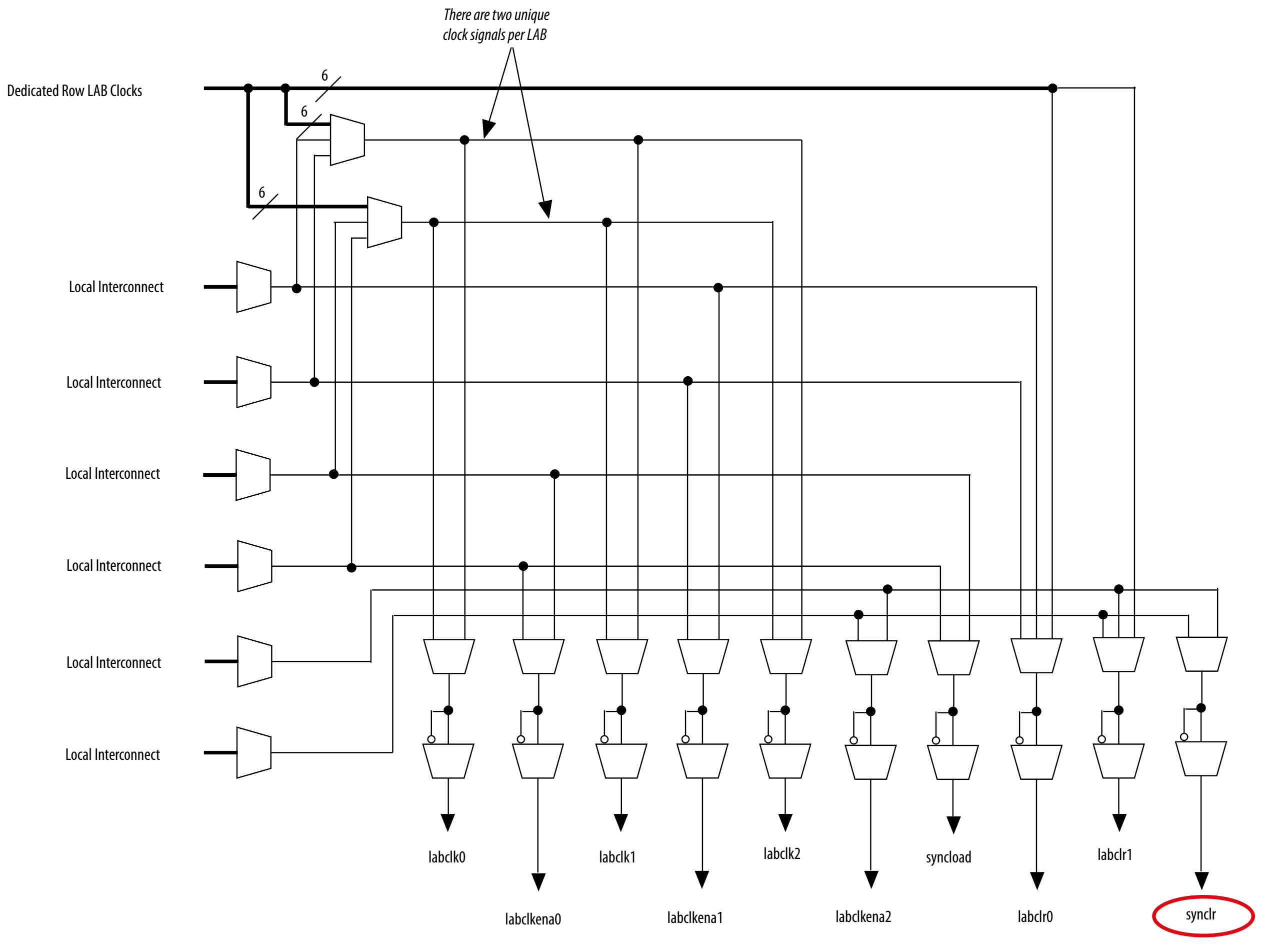
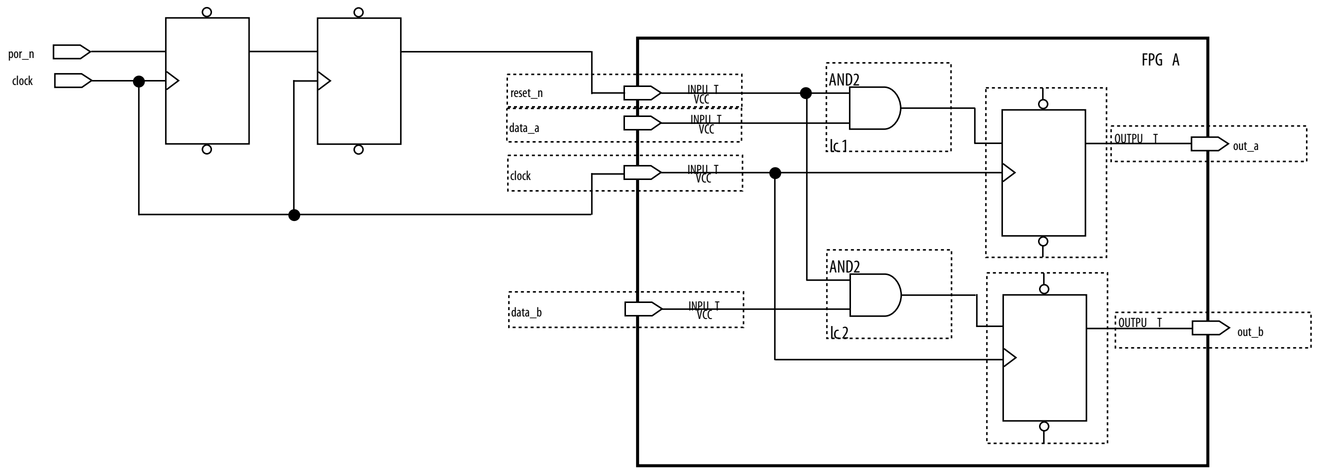


Figure 11-15: LAB-Wide Control Signals



Consider two types of synchronous resets when you examine the timing analysis of synchronous resets—externally synchronized resets and internally synchronized resets. Externally synchronized resets are synchronized to the clock domain outside the FPGA, and are not very common. A power-on asynchronous reset is dual-rank synchronized externally to the system clock and then brought into the FPGA. Inside the FPGA, gate this reset with the data input to the registers to implement a synchronous reset.

Figure 11-16: Externally Synchronized Reset



The following example shows the Verilog equivalent of the schematic. When you use synchronous resets, the reset signal is not put in the sensitivity list.

The following example shows the necessary modifications that you should make to the internally synchronized reset.

Example 11-6: Verilog Code for Externally Synchronized Reset

```
module sync_reset_ext (
         input   clock,
         input   reset_n,
         input   data_a,
         input   data_b,
         output  out_a,
         output  out_b
         );
reg      reg1, reg2
assign   out_a  = reg1;
assign   out_b  = reg2;
always @ (posedge clock)
begin
     if (!reset_n)
     begin
         reg1     <= 1’b0;
         reg2     <= 1’b0;
    end
    else
    begin
         reg1     <= data_a;
         reg2     <= data_b;
    end
end
endmodule     //  sync_reset_ext
```

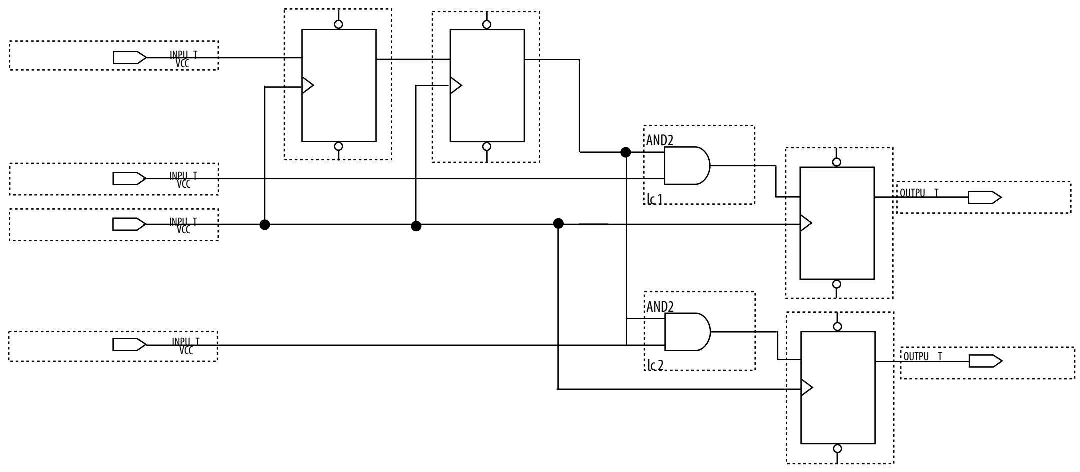
The following example shows the constraints for the externally synchronous reset. Because the external reset is synchronous, you only need to constrain the `reset_n` signal as a normal input signal with `set_input_delay` constraint for `-max` and `-min`.

Example 11-7: SDC Constraints for Externally Synchronized Reset

```
# Input clock - 100 MHz
create_clock [get_ports {clock}] \
        -name {clock} \
        -period 10.0 \
        -waveform {0.0 5.0}
# Input constraints on low-active reset
# and data
set_input_delay 7.0 \
        -max \
        -clock [get_clocks {clock}] \
        [get_ports {reset_n data_a data_b}]
set_input_delay 1.0 \
        -min \
        -clock [get_clocks {clock}] \
        [get_ports {reset_n data_a data_b}]
```

More often, resets coming into the device are asynchronous, and must be synchronized internally before being sent to the registers.

Figure 11-17: Internally Synchronized Reset



The following example shows the Verilog equivalent of the schematic. Only the clock edge is in the sensitivity list for a synchronous reset.

Example 11-8: Verilog Code for Internally Synchronized Reset

```
module sync_reset_ext (
         input   clock,
         input   reset_n,
         input   data_a,
         input   data_b,
         output  out_a,
         output  out_b
         );
reg      reg1, reg2
assign   out_a  = reg1;
assign   out_b  = reg2;
always @ (posedge clock)
begin
     if (!reset_n)
     begin
         reg1     <= 1’b0;
         reg2     <= 1’b0;
    end
    else
    begin
         reg1     <= data_a;
         reg2     <= data_b;
    end
end
endmodule     //  sync_reset_ext
```

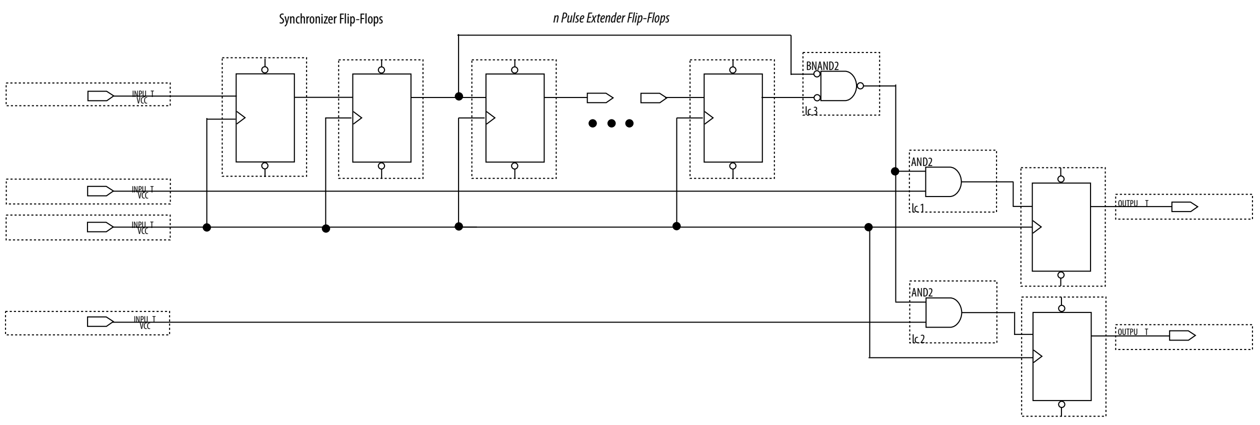
The SDC constraints are similar to the external synchronous reset, except that the input reset cannot be constrained because it is asynchronous and should be cut with a `set_false_path` statement to avoid these being considered as unconstrained paths.

Example 11-9: SDC Constraints for Internally Synchronized Reset

```
# Input clock - 100 MHz
create_clock [get_ports {clock}] \
        -name {clock} \
        -period 10.0 \
        -waveform {0.0 5.0}
# Input constraints on data
set_input_delay 7.0 \
        -max \
        -clock [get_clocks {clock}] \
        [get_ports {data_a data_b}]
set_input_delay 1.0 \
        -min \
        -clock [get_clocks {clock}] \
        [get_ports {data_a data_b}]
# Cut the asynchronous reset input
set_false_path \
        -from [get_ports {reset_n}] \
        -to [all_registers]
```

An issue with synchronous resets is their behavior with respect to short pulses (less than a period) on the asynchronous input to the synchronizer flipflops. This can be a disadvantage because the asynchronous reset requires a pulse width of at least one period wide to guarantee that it is captured by the first flipflop. However, this can also be viewed as an advantage in that this circuit increases noise immunity. Spurious pulses on the asynchronous input have a lower chance of being captured by the first flipflop, so the pulses do not trigger a synchronous reset. In some cases, you might want to increase the noise immunity further and reject any asynchronous input reset that is less than `n` periods wide to debounce an asynchronous input reset.

Figure 11-18: Internally Synchronized Reset with Pulse Extender



1. Junction dots indicate the number of stages. You can have more flip flops to get a wider pulse that spans more clock cycles.

   Many designs have more than one clock signal. In these cases, use a separate reset synchronization circuit for each clock domain in the design. When you create synchronizers for PLL output clocks, these clock domains are not reset until you lock the PLL and the PLL output clocks are stable. If you use the reset to the PLL, this reset does not have to be synchronous with the input clock of the PLL. You can use an asynchronous reset for this. Using a reset to the PLL further delays the assertion of a synchronous reset to the PLL output clock domains when using internally synchronized resets.

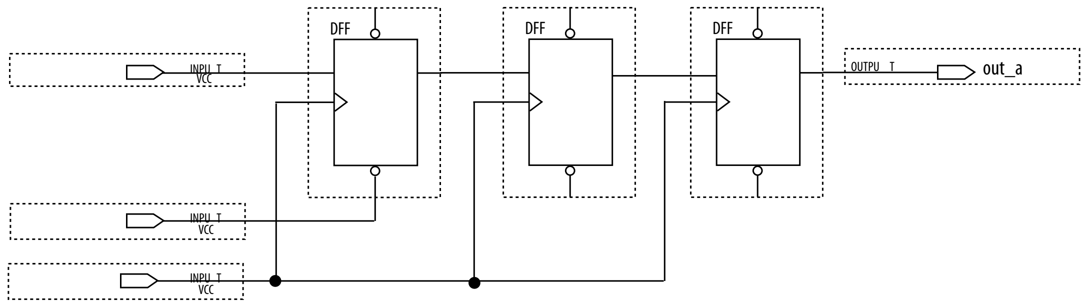
Using Asynchronous Resets

Asynchronous resets are the most common form of reset in circuit designs, as well as the easiest to implement. Typically, you can insert the asynchronous reset into the device, turn on the global buffer, and connect to the asynchronous reset pin of every register in the device.

This method is only advantageous under certain circumstances—you do not need to always reset the register. Unlike the synchronous reset, the asynchronous reset is not inserted in the data path, and does not negatively impact the data arrival times between registers. Reset takes effect immediately, and as soon as the registers receive the reset pulse, the registers are reset. The asynchronous reset is not dependent on the clock.

However, when the reset is deasserted and does not pass the recovery (μt_{SU}) or removal (μt_H) time check (the TimeQuest analyzer recovery and removal analysis checks both times), the edge is said to have fallen into the metastability zone. Additional time is required to determine the correct state, and the delay can cause the setup time to fail to register downstream, leading to system failure. To avoid this, add a few follower registers after the register with the asynchronous reset and use the output of these registers in the design. Use the follower registers to synchronize the data to the clock to remove the metastability issues. You should place these registers close to each other in the device to keep the routing delays to a minimum, which decreases data arrival times and increases MTBF. Ensure that these follower registers themselves are not reset, but are initialized over a period of several clock cycles by “flushing out” their current or initial state.

Figure 11-19: Asynchronous Reset with Follower Registers



The following example shows the equivalent Verilog code. The active edge of the reset is now in the sensitivity list for the procedural block, which infers a clock enable on the follower registers with the inverse of the reset signal tied to the clock enable. The follower registers should be in a separate procedural block as shown using non-blocking assignments.

Example 11-10: Verilog Code of Asynchronous Reset with Follower Registers

```
module async_reset (
        input   clock,
        input   reset_n,
        input   data_a,
        output   out_a,
        );
reg     reg1, reg2, reg3;
assign  out_a  = reg3;
always @ (posedge clock, negedge reset_n)
begin
    if (!reset_n)
        reg1    <= 1'b0;
    else
        reg1    <= data_a;
end
always @ (posedge clock)
begin
    reg2    <= reg1;
    reg3    <= reg2;
end
endmodule  //  async_reset
```

You can easily constrain an asynchronous reset. By definition, asynchronous resets have a non-deterministic relationship to the clock domains of the registers they are resetting. Therefore, static timing analysis of these resets is not possible and you can use the `set_false_path` command to exclude the path from timing analysis. Because the relationship of the reset to the clock at the register is not known, you cannot run recovery and removal analysis in the TimeQuest analyzer for this path. Attempting to do so even without the false path statement results in no paths reported for recovery and removal.

Example 11-11: SDC Constraints for Asynchronous Reset

```
# Input clock - 100 MHz
create_clock [get_ports {clock}] \
        -name {clock} \
        -period 10.0 \
        -waveform {0.0 5.0}
# Input constraints on data
set_input_delay 7.0 \
        -max \
        -clock [get_clocks {clock}]\
        [get_ports {data_a}]
set_input_delay 1.0 \
        -min \
        -clock [get_clocks {clock}] \
        [get_ports {data_a}]
# Cut the asynchronous reset input
set_false_path \
        -from [get_ports {reset_n}] \
        -to [all_registers]
```

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

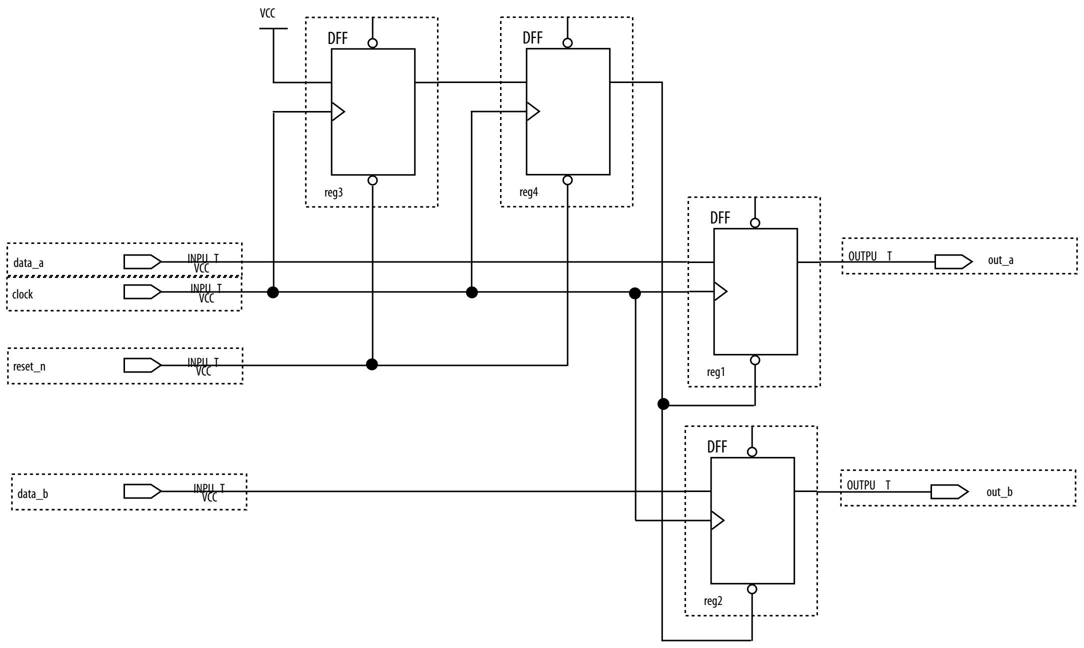
Use Synchronized Asynchronous Reset

To avoid potential problems associated with purely synchronous resets and purely asynchronous resets, you can use synchronized asynchronous resets. Synchronized asynchronous resets combine the advantages of synchronous and asynchronous resets.

These resets are asynchronously asserted and synchronously deasserted. This takes effect almost instantaneously, and ensures that no data path for speed is involved, and that the circuit is synchronous for timing analysis and is resistant to noise.

The following example shows a method for implementing the synchronized asynchronous reset. You should use synchronizer registers in a similar manner as synchronous resets. However, the asynchronous reset input is gated directly to the `CLRN` pin of the synchronizer registers and immediately asserts the resulting reset. When the reset is deasserted, logic “1” is clocked through the synchronizers to synchronously deassert the resulting reset.

Figure 11-20: Schematic of Synchronized Asynchronous Reset



The following example shows the equivalent Verilog HDL code. Use the active edge of the reset in the sensitivity list for the blocks.

Example 11-12: Verilog Code for Synchronized Asynchronous Reset

```
module sync_async_reset (
        input    clock,
        input    reset_n,
        input    data_a,
        input    data_b,
        output   out_a,
        output   out_b
        );
reg     reg1, reg2;
reg     reg3, reg4;
assign  out_a    = reg1;
assign  out_b    = reg2;
assign  rst_n    = reg4;
always @ (posedge clock, negedge reset_n)
begin
    if (!reset_n)
    begin
       reg3     <= 1’b0;
       reg4     <= 1’b0;
    end
    else
    begin
       reg3     <= 1’b1;
       reg4     <= reg3;
    end
end
always @ (posedge clock, negedge rst_n)
begin
    if (!rst_n)
```

```
        begin
           reg1     <= 1’b0;
           reg2     <= 1;b0;
        end
        else
        begin
           reg1     <= data_a;
           reg2     <= data_b;
        end
    end
    endmodule  // sync_async_reset
```

To minimize the metastability effect between the two synchronization registers, and to increase the MTBF, the registers should be located as close as possible in the device to minimize routing delay. If possible, locate the registers in the same logic array block (LAB). The input reset signal (`reset_n`) must be excluded with a `set_false_path` command:

```
set_false_path -from [get_ports {reset_n}] -to [all_registers]
```

The `set_false_path` command used with the specified constraint excludes unnecessary input timing reports that would otherwise result from specifying an input delay on the reset pin.

The instantaneous assertion of synchronized asynchronous resets is susceptible to noise and runt pulses. If possible, you should debounce the asynchronous reset and filter the reset before it enters the device. The circuit ensures that the synchronized asynchronous reset is at least one full clock period in length. To extend this time to n clock periods, you must increase the number of synchronizer registers to n + 1. You must connect the asynchronous input reset (`reset_n`) to the `CLRN` pin of all the synchronizer registers to maintain the asynchronous assertion of the synchronized asynchronous reset.

Avoid Asynchronous Register Control Signals

Avoid using an asynchronous load signal if the design target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of these control signals.

Some Altera devices directly support an asynchronous clear function, but not a preset or load function. When the target device does not directly support the signals, the synthesis or placement and routing software must use combinational logic to implement the same functionality. In addition, if you use signals in a priority other than the inherent priority in the device architecture, combinational logic may be required to implement the necessary control signals. Combinational logic is less efficient and can cause glitches and other problems; it is best to avoid these implementations.

Implementing Embedded RAM

Altera’s dedicated memory architecture offers many advanced features that you can enable with Altera-provided IP cores. Use synchronous memory blocks for your design, so that the blocks can be mapped directly into the device dedicated memory blocks.

You can use single-port, dual-port, or three-port RAM with a single- or dual-clocking method. You should not infer the asynchronous memory logic as a memory block or place the asynchronous memory logic in the dedicated memory block, but implement the asynchronous memory logic in regular logic cells.

Altera memory blocks have different read-during-write behaviors, depending on the targeted device family, memory mode, and block type. Read-during-write behavior refers to read and write from the same memory address in the same clock cycle; for example, you read from the same address to which you write in the same clock cycle.

You should check how you specify the memory in your HDL code when you use read-during-write behavior. The HDL code that describes the read returns either the old data stored at the memory location, or the new data being written to the memory location.

In some cases, when the device architecture cannot implement the memory behavior described in your HDL code, the memory block is not mapped to the dedicated RAM blocks, or the memory block is implemented using extra logic in addition to the dedicated RAM block. Implement the read-during-write behavior using single-port RAM in Arria GX devices and the Cyclone and Stratix series of devices to avoid this extra logic implementation.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; if, for example, you never read and write from the same address in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the read-during-write behavior specified in your HDL code. Using this type of attribute prevents the synthesis tool from using extra logic to implement the memory block and, in some cases, can allow memory inference when it would otherwise be impossible.

Document Revision History

Table 11-1: Document Revision History

Date	Version	Changes
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.
June 2014	14.0.0	Removed references to obsolete MegaWizard Plug-In Manager.
November 2013	13.1.0	Removed HardCopy device information.
May 2013	13.0.0	Removed PrimeTime support.
June 2012	12.0.0	Removed survey link.
November 2011	11.0.1	Template update.
May 2011	11.0.0	Added information to Reset Resources .
December 2010	10.1.0	• Title changed from Design Recommendations for Altera Devices and the Quartus II Design Assistant. • Updated to new template. • Added references to Quartus II Help for "Metastability" on page 9–13 and "Incremental Compilation" on page 9–13. • Removed duplicated content and added references to Quartus II Help for "Custom Rules" on page 9–15.

Date	Version	Changes
July 2010	10.0.0	• Removed duplicated content and added references to Quartus II Help for Design Assistant settings, Design Assistant rules, Enabling and Disabling Design Assistant Rules, and Viewing Design Assistant reports. • Removed information from "Combinational Logic Structures" on page 5–4 • Changed heading from "Design Techniques to Save Power" to "Power Optimization" on page 5–12 • Added new "Metastability" section • Added new "Incremental Compilation" section • Added information to "Reset Resources" on page 5–23 • Removed "Referenced Documents" section
November 2009	9.1.0	• Removed documentation of obsolete rules.
March 2009	9.0.0	• No change to content.
November 2008	8.1.0	• Changed to 8-1/2 x 11 page size • Added new section "Custom Rules Coding Examples" on page 5–18 • Added paragraph to "Recommended Clock-Gating Methods" on page 5–11 • Added new section: "Design Techniques to Save Power" on page 5–12
May 2008	8.0.0	• Updated Figure 5–9 on page 5–13; added custom rules file to the flow • Added notes to Figure 5–9 on page 5–13 • Added new section: "Custom Rules Report" on page 5–34 • Added new section: "Custom Rules" on page 5–34 • Added new section: "Targeting Embedded RAM Architectural Features" on page 5–38 • Minor editorial updates throughout the chapter • Added hyperlinks to referenced documents throughout the chapter

Related Information

http://www.altera.com/literature/lit-qts_archive.jsp

Recommended HDL Coding Styles 12

2015.05.04

QII5V1 Subscribe Send Feedback

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

HDL coding styles can have a significant effect on the quality of results that you achieve for programmable logic designs. Synthesis tools optimize HDL code for both logic utilization and performance; however, synthesis tools have no information about the purpose or intent of the design. The best optimizations require your conscious interaction. The Altera website provides design examples for other types of functions and to target specific applications.

Note: For style recommendations, options, or HDL attributes specific to your synthesis tool (including Quartus II integrated synthesis and other EDA tools), refer to the tool vendor's documentation.

Related Information

- Recommended Design Practices on page 11-1
- Advanced Synthesis Cookbook
- Design Examples
- Reference Designs
- Quartus II Integrated Synthesis on page 16-1

Using Provided HDL Templates

You can use provided HDL templates to start your HDL designs.

Altera provides templates for Verilog HDL, SystemVerilog, and VHDL. Many of the HDL examples in this document correspond with the**Full Designs** examples in the **Quartus II Templates**. You can insert HDL code into your own design using the templates or examples.

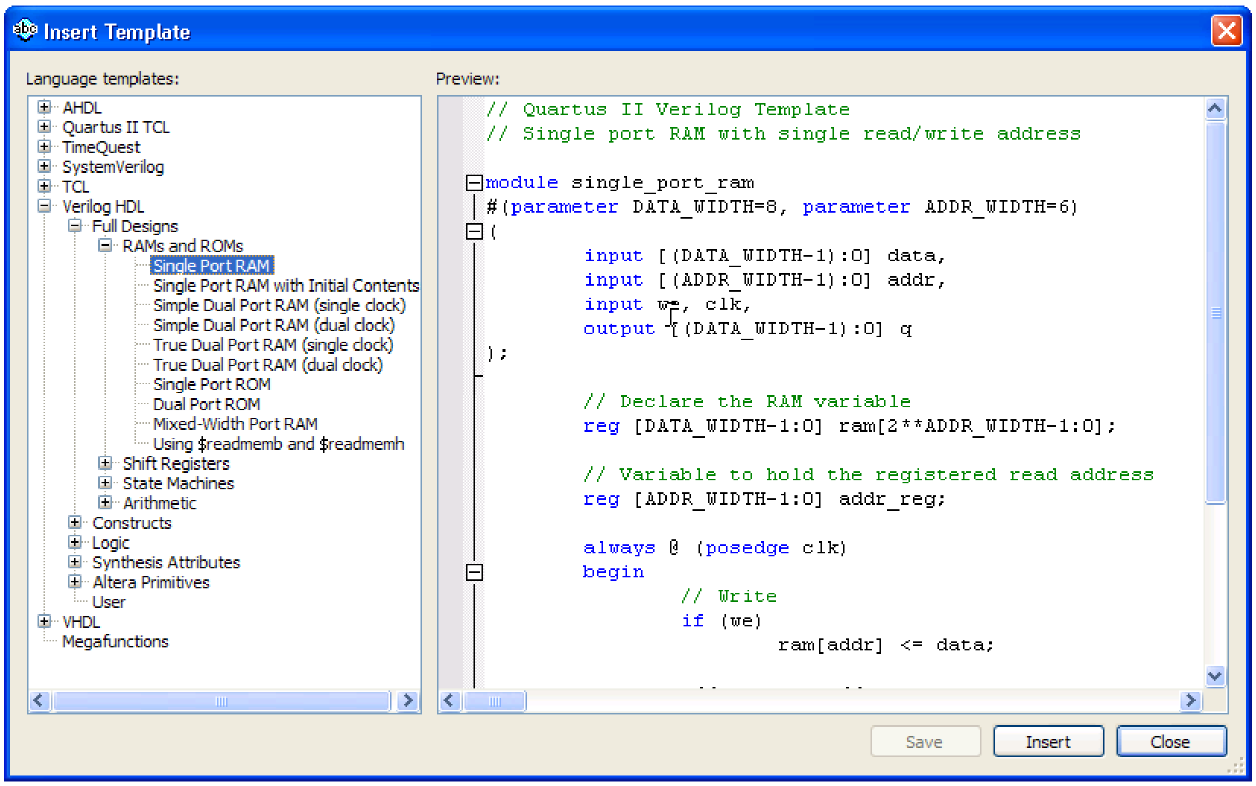
Inserting a HDL Code from the Template

Insert HDL code from a provided template, follow these steps:

1. On the **File** menu, click **New**.
2. In the **New** dialog box, select the type of design file corresponding to the type of HDL you want to use, SystemVerilog HDL File, VHDL File, or Verilog HDL File.
3. Right-click in the HDL file and then click **Insert Template**.
4. In the **Insert Template** dialog box, expand the section corresponding to the appropriate HDL, then expand the **Full Designs** section.

5. Select a design. The HDL appears in the **Preview** pane.
6. Click **Insert** to paste the HDL design to the blank Verilog or VHDL file you created in step 2.
7. Click **Close** to close the **Insert Template** dialog box.

Figure 12-1: Inserting a RAM Template



AHDL
Quartus II TCL
TimeQuest
SystemVerilog
TCL
Verilog HDL
- Full Designs
  - RAMs and ROMs
    - Single Port RAM
    - Single Port RAM with Initial Contents
    - Simple Dual Port RAM (single clock)
    - Simple Dual Port RAM (dual clock)
    - True Dual Port RAM (single clock)
    - True Dual Port RAM (dual clock)
    - Single Port ROM
    - Dual Port ROM
    - Mixed-Width Port RAM
    - Using $readmemb and $readmemh
  - Shift Registers
  - State Machines
  - Arithmetic
- Constructs
- Logic
- Synthesis Attributes
- Altera Primitives
- User
VHDL
Megafunctions

```
// Quartus II Verilog Template
// Single port RAM with single read/write address

module single_port_ram
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input we, clk,
        output [(DATA_WIDTH-1):0] q
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        // Variable to hold the registered read address
        reg [ADDR_WIDTH-1:0] addr_reg;

        always @ (posedge clk)
        begin
                // Write
                if (we)
                        ram[addr] <= data;
```

Note: You can use any of the standard features of the Quartus II Text Editor to modify the HDL design or save the template as an HDL file to edit in your preferred text editor.

Related Information

About the Quartus II Text Editor

Instantiating IP Cores in HDL

Altera provides parameterizable IP cores that are optimized for Altera device architectures. Using IP cores instead of coding your own logic saves valuable design time.

Additionally, the Altera-provided IP cores offer more efficient logic synthesis and device implementation. You can scale the IP core's size and specify various options by setting parameters. You can instantiate the IP core directly in your HDL file code by calling the IP core name and defining its parameters as you would any other module, component, or subdesign. Alternatively, you can use the IP Catalog (**Tools** > **IP**

Catalog) and parameter editor GUI to simplify customization of your IP core variation. You can infer or instantiate IP cores that optimize the following device architecture features:

- Transceivers
- LVDS drivers
- Memory and DSP blocks
- Phase-locked loops (PLLs)
- double-data rate input/output (DDIO) circuitry

For some types of logic functions, such as memories and DSP functions, you can infer device-specific dedicated architecture blocks instead of instantiating an IP core. Quartus II synthesis recognizes certain HDL code structures and automatically infers the appropriate IP core or map directly to device atoms.

Related Information

- **Inferring Multipliers and DSP Functions** on page 12-3
- **Inferring Memory Functions from HDL Code** on page 12-8
- **Altera IP Core Literature**

Inferring Multipliers and DSP Functions

The following sections describe how to infer multiplier and DSP functions from generic HDL code, and, if applicable, how to target the dedicated DSP block architecture in Altera devices.

Related Information

DSP Solutions Center

Inferring Multipliers

To infer multiplier functions, synthesis tools detect multiplier logic and implement this in Altera IP cores, or map the logic directly to device atoms.

For devices with DSP blocks, the software can implement the function in a DSP block instead of logic, depending on device utilization. The Quartus II Fitter can also place input and output registers in DSP blocks (that is, perform register packing) to improve performance and area utilization.

The Verilog HDL and VHDL code examples show, for unsigned and signed multipliers, that synthesis tools can infer as an IP core or DSP block atoms. Each example fits into one DSP block element. In addition, when register packing occurs, no extra logic cells for registers are required.

Note: The `signed` declaration in Verilog HDL is a feature of the Verilog 2001 Standard.

Example 12-1: Verilog HDL Unsigned Multiplier

```
module unsigned_mult (out, a, b);
    output [15:0] out;
    input [7:0] a;
    input [7:0] b;
    assign out = a * b;
endmodule
```

Example 12-2: Verilog HDL Signed Multiplier with Input and Output Registers (Pipelining = 2)

```
module signed_mult (out, clk, a, b);
   output [15:0] out;
   input clk;
   input signed [7:0] a;
   input signed [7:0] b;

   reg signed [7:0] a_reg;
   reg signed [7:0] b_reg;
   reg signed [15:0] out;
   wire signed [15:0] mult_out;

   assign mult_out = a_reg * b_reg;

   always @ (posedge clk)
   begin
      a_reg <= a;
      b_reg <= b;
      out <= mult_out;
   end
endmodule
```

Example 12-3: VHDL Unsigned Multiplier with Input and Output Registers (Pipelining = 2)

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

ENTITY unsigned_mult IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_reg, b_reg: UNSIGNED (7 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr ='1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         result <= (OTHERS => '0');
      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         result <= a_reg * b_reg;
      END IF;
   END PROCESS;
END rtl;
```

Example 12-4: VHDL Signed Multiplier

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

ENTITY signed_mult IS
   PORT (
      a: IN SIGNED (7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      result: OUT SIGNED (15 DOWNTO 0)
   );
END signed_mult;

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

Inferring Multiply-Accumulator and Multiply-Adder

Synthesis tools detect multiply-accumulate or multiply-add functions and implement them as Altera IP cores, respectively, or may map them directly to device atoms. The Quartus II software then places these functions in DSP blocks during placement and routing.

Note: Synthesis tools infer multiply-accumulator and multiply-adder functions only if the Altera device family has dedicated DSP blocks that support these functions.

A simple multiply-accumulator consists of a multiplier feeding an addition operator. The addition operator feeds a set of registers that then feeds the second input to the addition operator. A simple multiply-adder consists of two to four multipliers feeding one or two levels of addition, subtraction, or addition/subtraction operators. Addition is always the second-level operator, if it is used. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter also places input and output registers into the DSP blocks to pack registers and improve performance and area utilization.

Some device families offer additional advanced multiply-add and accumulate functions, such as complex multiplication, input shift register, or larger multiplications.

The Verilog HDL and VHDL code samples infer multiply-accumulators and multiply-adders with input, output, and pipeline registers, as well as an optional asynchronous clear signal. Using the three sets of registers provides the best performance through the function, with a latency of three. You can remove the registers in your design to reduce the latency.

Note: To obtain high performance in DSP designs, use register pipelining and avoid unregistered DSP functions.

Example 12-5: Verilog HDL Unsigned Multiply-Accumulator

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

   reg [7:0] dataa_reg, datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [16:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;

   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 8'b0;
         datab_reg <= 8'b0;
         multa_reg <= 16'b0;
         dataout <= 17'b0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

Example 12-6: Verilog HDL Signed Multiply-Adder

```
module sig_altmult_add (dataa, datab, datac, datad, clock, aclr, result);
   input signed [15:0] dataa, datab, datac, datad;
   input clock, aclr;
   output reg signed [32:0] result;

   reg signed [15:0] dataa_reg, datab_reg, datac_reg, datad_reg;
   reg signed [31:0] mult0_result, mult1_result;

   always @ (posedge clock or posedge aclr) begin
       if (aclr) begin
           dataa_reg <= 16'b0;
           datab_reg <= 16'b0;
           datac_reg <= 16'b0;
           datad_reg <= 16'b0;
           mult0_result <= 32'b0;
           mult1_result <= 32'b0;
           result <= 33'b0;
        end
        else begin
           dataa_reg <= dataa;
           datab_reg <= datab;
           datac_reg <= datac;
           datad_reg <= datad;
           mult0_result <= dataa_reg * datab_reg;
           mult1_result <= datac_reg * datad_reg;
           result <= mult0_result + mult1_result;
       end
   end
endmodule
```

Example 12-7: VHDL Signed Multiply-Accumulator

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY sig_altmult_accum IS
   PORT (
      a: IN SIGNED(7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      accum_out: OUT SIGNED (15 DOWNTO 0)
   ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
   SIGNAL a_reg, b_reg: SIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg: SIGNED (15 DOWNTO 0);
   SIGNAL adder_out: SIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') then
          a_reg <= (others => '0');
          b_reg <= (others => '0');
          pdt_reg <= (others => '0');
          adder_out <= (others => '0');
      ELSIF (clk'event and clk = '1') THEN
          a_reg <= (a);
          b_reg <= (b);
          pdt_reg <= a_reg * b_reg;
          adder_out <= adder_out + pdt_reg;
      END IF;
   END process;
   accum_out <= adder_out;
END rtl;
```

Example 12-8: VHDL Unsigned Multiply-Adder

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

ENTITY unsignedmult_add IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      c: IN UNSIGNED (7 DOWNTO 0);
      d: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
   SIGNAL a_reg, b_reg, c_reg, d_reg: UNSIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg, pdt2_reg: UNSIGNED (15 DOWNTO 0);
   SIGNAL result_reg: UNSIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
```

```
            c_reg <= (OTHERS => '0');
            d_reg <= (OTHERS => '0');
            pdt_reg <= (OTHERS => '0');
            pdt2_reg <= (OTHERS => '0');

         ELSIF (clk'event AND clk = '1') THEN
            a_reg <= a;
            b_reg <= b;
            c_reg <= c;
            d_reg <= d;
            pdt_reg <= a_reg * b_reg;
            pdt2_reg <= c_reg * d_reg;
            result_reg <= pdt_reg + pdt2_reg;
         END IF;
         END PROCESS;
      result <= result_reg;
   END rtl;
```

Related Information

- DSP Design Examples
- AN639: Inferring Stratix V DSP Blocks for FIR Filtering

Inferring Memory Functions from HDL Code

The following sections describe how to infer memory functions and target dedicated memory architecture using HDL code.

Altera's dedicated memory architecture offers a number of advanced features that can be easily targeted by instantiating Altera various Altera memory IP Cores in HDL. The following coding recommendations provide portable examples of generic HDL code that infer the appropriate Altera memory IP core. However, if you want to use some of the advanced memory features in Altera devices, consider using the IP core directly so that you can customize the ports and parameters easily. You can also use the Quartus II templates provided in the Quartus II software as a starting point.

Most of these designs can also be found on the Design Examples page on the Altera website.

Table 12-1: Altera Memory HDL Design Examples

Language	Full Design Name
VHDL	Single-Port RAM Single-Port RAM with Initial Contents Simple Dual-Port RAM (single clock)Simple Dual-Port RAM (dual clock) True Dual-Port RAM (single clock) True Dual-Port RAM (dual clock) Mixed-Width RAM Mixed-Width True Dual-Port RAM Byte-Enabled Simple Dual-Port RAM Byte-Enabled True Dual-Port RAM Single-Port ROMDual-Port ROM
Verilog HDL	Single-Port RAM Single-Port RAM with Initial Contents Simple Dual-Port RAM (single clock) Simple Dual-Port RAM (dual clock) True Dual-Port RAM (single clock) True Dual-Port RAM (dual clock) Single-Port ROM Dual-Port ROM
System Verilog	Mixed-Width Port RAM Mixed-Width True Dual-Port RAM Mixed-Width True Dual-Port RAM (new data on same port read during write) Byte-Enabled Simple Dual Port RAM Byte-Enabled True Dual-Port RAM

Related Information

- Instantiating Altera IP Cores in HDL Code
- Design Examples

Inferring RAM functions from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with Altera IP cores for device families that have dedicated RAM blocks, or may map them directly to device memory atoms.

Synthesis tools typically consider all signals and variables that have a multi-dimensional array type and then create a RAM block, if applicable. This is based on the way the signals or variables are assigned or referenced in the HDL source description.

Standard synthesis tools recognize single-port and simple dual-port (one read port and one write port) RAM blocks. Some tools (such as the Quartus II software) also recognize true dual-port (two read ports and two write ports) RAM blocks that map to the memory blocks in certain Altera devices.

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

Note: If your design contains a RAM block that your synthesis tool does not recognize and infer, the design might require a large amount of system memory that can potentially cause compilation problems

When you use a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules that contain only the RAM logic. In certain formal verification flows, for example, when using Quartus II integrated synthesis, the entity or module containing the inferred RAM is put into a black box automatically because formal verification tools do not support RAM blocks. The Quartus II software issues a warning message when this situation occurs. If the entity or module contains any additional logic outside the RAM block, this logic cannot be verified because it also must be treated as a black box for formal verification.

Use Synchronous Memory Blocks

Use synchronous memory blocks for Altera designs.

Because memory blocks in the newest devices from Altera are synchronous, RAM designs that are targeted towards architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture. For these devices, asynchronous memory logic is implemented in regular logic cells.

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

Synchronous memories are supported in all Altera device families. A memory block is considered synchronous if it uses one of the following read behaviors:

- Memory read occurs in a Verilog `always` block with a clock signal or a VHDL clocked process. The recommended coding style for synchronous memories is to create your design with a registered read output.
- Memory read occurs outside a clocked block, but there is a synchronous read address (that is, the address used in the read statement is registered). This type of logic is not always inferred as a memory block, or may require external bypass logic, depending on the target device architecture.

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

Later sections provide coding recommendations for various memory types. All of these examples are synchronous to ensure that they can be directly mapped into the dedicated memory architecture available in Altera FPGAs.

Avoid Unsupported Reset and Control Conditions

To ensure that your HDL code can be implemented in the target device architecture, avoid unsupported reset conditions or other control logic that does not exist in the device architecture.

The RAM contents of Altera memory blocks cannot be cleared with a reset signal during device operation. If your HDL code describes a RAM with a reset signal for the RAM contents, the logic is implemented in regular logic cells instead of a memory block. Altera recommends against putting RAM read or write operations in an `always` block or `process` block with a reset signal. If you want to specify memory contents, initialize the memory or write the data to the RAM during device operation.

In addition to reset signals, other control logic can prevent memory logic from being inferred as a memory block. For example, you cannot use a clock enable on the read address registers in some devices because this affects the output latch of the RAM, and therefore the synthesized result in the device RAM architecture would not match the HDL description. You can use the address stall feature as a read address clock enable to avoid this limitation. Check the documentation for your device architecture to ensure that your code matches the hardware available in the device.

Example 12-9: Verilog RAM with Reset Signal that Clears RAM Contents: Not Supported in Device Architecture

```
module clear_ram
(
    input clock, reset, we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            mem[address] <= 0;
        else if (we == 1'b1)
            mem[address] <= data_in;

        data_out <= mem[address];
    end
endmodule
```

Example 12-10: Verilog RAM with Reset Signal that Affects RAM: Not Supported in Device Architecture

```
module bad_reset
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out,
    input d,
    output reg q
);
```

```
    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            q <= 0;
        else
        begin
            if (we == 1'b1)
                mem[address] <= data_in;

            data_out <= mem[address];
            q <= d;
        end
    end
endmodule
```

Related Information

Specifying Initial Memory Contents at Power-Up on page 12-25

Check Read-During-Write Behavior

It is important to check the read-during-write behavior of the memory block described in your HDL design as compared to the behavior in your target device architecture.

Your HDL source code specifies the memory behavior when you read and write from the same memory address in the same clock cycle. The code specifies that the read returns either the old data at the address, or the new data being written to the address. This behavior is referred to as the read-during-write behavior of the memory block. Altera memory blocks have different read-during-write behavior depending on the target device family, memory mode, and block type.

Synthesis tools map an HDL design into the target device architecture, with the goal of maintaining the functionality described in your source code. Therefore, if your source code specifies unsupported read-during-write behavior for the device RAM blocks, the software must implement the logic outside the RAM hardware in regular logic cells.

One common problem occurs when there is a continuous read in the HDL code, as in the following examples. You should avoid using these coding styles:

```
//Verilog HDL concurrent signal assignment
assign q = ram[raddr_reg];

-- VHDL concurrent signal assignment
q <= ram(raddr_reg);
```

When a write operation occurs, this type of HDL implies that the read should immediately reflect the new data at the address, independent of the read clock. However, that is not the behavior of synchronous memory blocks. In the device architecture, the new data is not available until the next edge of the read clock. Therefore, if the synthesis tool mapped the logic directly to a synchronous memory block, the device functionality and gate-level simulation results would not match the HDL description or functional simulation results. If the write clock and read clock are the same, the synthesis tool can infer memory blocks and add extra bypass logic so that the device behavior matches the HDL behavior. If the write and read clocks are different, the synthesis tool cannot reliably add bypass logic, so the logic is implemented in regular logic cells instead of dedicated RAM blocks. The examples in the following sections discuss some of these differences for read-during-write conditions.

In addition, the MLAB feature in certain device logic array blocks (LABs) does not easily support old data or new data behavior for a read-during-write in the dedicated device architecture. Implementing the extra logic to support this behavior significantly reduces timing performance through the memory.

Note: For best performance in MLAB memories, your design should not depend on the read data during a write operation.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; for example, if you never read from the same address to which you write in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle` set to `"no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code. In some cases, this attribute prevents the synthesis tool from using extra logic to implement the memory block, or can allow memory inference when it would otherwise be impossible.

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

Related Information

- **Quartus II Integrated Synthesis** on page 16-1

Controlling RAM Inference and Implementation

Synthesis tools usually do not infer small RAM blocks because small RAM blocks typically can be implemented more efficiently using the registers in regular logic.

If you are using Quartus II integrated synthesis, you can direct the software to infer RAM blocks for all sizes with the **Allow Any RAM Size for Recognition** option in the **Advanced Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with synchronous memory blocks. For example, Quartus II integrated synthesis provides the `ramstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block. Quartus II integrated synthesis does not map inferred memory into MLABs unless the HDL code specifies the appropriate `ramstyle` attribute, although the Fitter may map some memories to MLABs.

If you want to control the implementation after the RAM function is inferred during synthesis, you can set the `ram_block_type` parameter of the ALTSYNCRAM IP core. In the Assignment Editor, select **Parameters** in the **Categories** list. You can use the **Node Finder** or drag the appropriate instance from the Project Navigator window to enter the RAM hierarchical instance name. Type `ram_block_type` as the **Parameter Name** and type one of the following memory types supported by your target device family in the **Value** field: "`M-RAM`", "`M512`", "`M4K`", "`M9K`", "`M10K`", "`M20K`", "`M144K`", or "`MLAB`".

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

```
// Limit the depth of the memory blocks implement "ram" to 512
// This forces the software to use two M512 blocks instead of one M4K block to
implement this RAM
(* max_depth = 512 *) reg [7:0] ram[0:1023];
```

Related Information

- **Quartus II Integrated Synthesis** on page 16-1

Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior

The code examples in this section show Verilog HDL and VHDL code that infers simple dual-port, single-clock synchronous RAM. Single-port RAM blocks use a similar coding style.

The read-during-write behavior in these examples is to read the old data at the memory address. Altera recommends that you use the Old Data Read-During-Write coding style for most RAM blocks as long as your design does not require the RAM location's new value when you perform a simultaneous read and write to that RAM location. For best performance in MLAB memories, use the appropriate attribute so that your design does not depend on the read data during a write operation. The simple dual-port RAM code samples map directly into Altera synchronous memory.

Single-port versions of memory blocks (that is, using the same read address and write address signals) can allow better RAM utilization than dual-port memory blocks, depending on the device family.

Example 12-11: Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
module single_clk_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] <= d;
        q <= mem[read_address]; // q doesn't get d in this clock cycle
    end
endmodule
```

Example 12-12: VHDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

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

ENTITY single_clock_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_ram;

ARCHITECTURE rtl OF single_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
```

```
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
            -- VHDL semantics imply that q doesn't get data
            -- in this clock cycle
        END IF;
    END PROCESS;
END rtl;
```

Related Information

- Check Read-During-Write Behavior on page 12-12
- Single-Clock Synchronous RAM with New Data Read-During-Write Behavior on page 12-15

Single-Clock Synchronous RAM with New Data Read-During-Write Behavior

The examples in this section describe RAM blocks in which a simultaneous read and write to the same location reads the new value that is currently being written to that RAM location.

To implement this behavior in the target device, synthesis software adds bypass logic around the RAM block. This bypass logic increases the area utilization of the design and decreases the performance if the RAM block is part of the design's critical path.

Single-port versions of the Verilog memory block (that is, using the same read address and write address signals) do not require any logic cells to create bypass logic in the Arria, Stratix, and Cyclone series of devices, because the device memory supports new data read-during-write behavior when in single-port mode (same clock, same read address, and same write address).

For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the behavior specified by your HDL code. This attribute may prevent generation of extra bypass logic, but it is not always possible to eliminate the requirement for bypass logic.

Example 12-13: Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
module single_clock_wr_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] = d;
        q = mem[read_address]; // q does get d in this clock cycle
if                                // we is high
    end
endmodule
```

It is possible to create a single-clock RAM using an assign statement to read the address of `mem` to create the output q. By itself, the code describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, a read-during-write results in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary. Avoid this type of coding.

Example 12-14: Avoid This Coding Style

```
reg [7:0] mem [127:0];
reg [6:0] read_address_reg;

always @ (posedge clk) begin
    if (we)
        mem[write_address] <= d;

    read_address_reg <= read_address;
end

assign q = mem[read_address_reg];
```

The following example uses a concurrent signal assignment to read from the RAM. By itself, this example describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, a read-during-write results in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary

Example 12-15: VHDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

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

ENTITY single_clock_rw_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_rw_ram;

ARCHITECTURE rtl OF single_clock_rw_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
```

```
     END PROCESS;
     q <= ram_block(read_address_reg);
END rtl;
```

For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the behavior specified by your HDL code. This attribute may prevent generation of extra bypass logic but it is not always possible to eliminate the requirement for bypass logic.

Related Information

- **Check Read-During-Write Behavior** on page 12-12
- **Check Read-During-Write Behavior** on page 12-12
- **Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior** on page 12-14

Simple Dual-Port, Dual-Clock Synchronous RAM

In dual clock designs, synthesis tools cannot accurately infer the read-during-write behavior because it depends on the timing of the two clocks within the target device.

Therefore, the read-during-write behavior of the synthesized design is undefined and may differ from your original HDL code. When Quartus II integrated synthesis infers this type of RAM, it issues a warning because of the undefined read-during-write behavior.

Example 12-16: Verilog HDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
module dual_clock_ram(
   output reg [7:0] q,
   input [7:0] d,
   input [6:0] write_address, read_address,
   input we, clk1, clk2
);
   reg [6:0] read_address_reg;
   reg [7:0] mem [127:0];

   always @ (posedge clk1)
   begin
      if (we)
         mem[write_address] <= d;
   end

   always @ (posedge clk2) begin
      q <= mem[read_address_reg];
      read_address_reg <= read_address;
   end
endmodule
```

Example 12-17: VHDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dual_clock_ram IS
    PORT (
        clock1, clock2: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END dual_clock_ram;
ARCHITECTURE rtl OF dual_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg : INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock1)
    BEGIN
        IF (clock1'event AND clock1 = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
        END IF;
    END PROCESS;
    PROCESS (clock2)
    BEGIN
        IF (clock2'event AND clock2 = '1') THEN
            q <= ram_block(read_address_reg);
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
END rtl;
```

Related Information

Check Read-During-Write Behavior on page 12-12

True Dual-Port Synchronous RAM

The code examples in this section show Verilog HDL and VHDL code that infers true dual-port synchronous RAM. Different synthesis tools may differ in their support for these types of memories.

Altera synchronous memory blocks have two independent address ports, allowing for operations on two unique addresses simultaneously. A read operation and a write operation can share the same port if they share the same address. The Quartus II software infers true dual-port RAMs in Verilog HDL and VHDL with any combination of independent read or write operations in the same clock cycle, with at most two unique port addresses, performing two reads and one write, two writes and one read, or two writes and two reads in one clock cycle with one or two unique addresses.

In the synchronous RAM block architecture, there is no priority between the two ports. Therefore, if you write to the same location on both ports at the same time, the result is indeterminate in the device architecture. You must ensure your HDL code does not imply priority for writes to the memory block, if you want the design to be implemented in a dedicated hardware memory block. For example, if both ports are defined in the same process block, the code is synthesized and simulated sequentially so that there is a priority between the two ports. If your code does imply a priority, the logic cannot be implemented in the device RAM blocks and is implemented in regular logic cells. You must also consider the read-during-write behavior of the RAM block to ensure that it can be mapped directly to the device RAM architecture.

When a read and write operation occurs on the same port for the same address, the read operation may behave as follows:

- **Read new data**—This mode matches the behavior of synchronous memory blocks.
- **Read old data**—This mode is supported only in device families that support M144K and M9K memory blocks.

When a read and write operation occurs on different ports for the same address (also known as mixed port), the read operation may behave as follows:

- **Read new data**—Quartus II integrated synthesis supports this mode by creating bypass logic around the synchronous memory block.
- **Read old data**—Synchronous memory blocks support this behavior.
- **Read don't care**—This behavior is supported on different ports in simple dual-port mode by synchronous memory blocks.

The Verilog HDL single-clock code sample maps directly into Altera synchronous memory. When a read and write operation occurs on the same port for the same address, the new data being written to the memory is read. When a read and write operation occurs on different ports for the same address, the old data in the memory is read. Simultaneous writes to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 12-18: Verilog HDL True Dual-Port RAM with Single Clock

```
module true_dual_port_ram_single_clock
(
    input [(DATA_WIDTH-1):0] data_a, data_b,
    input [(ADDR_WIDTH-1):0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [(DATA_WIDTH-1):0] q_a, q_b
);

    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 6;

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin // Port A
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
            q_a <= ram[addr_a];
    end
    always @ (posedge clk)
    begin // Port b
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
            q_b <= ram[addr_b];
    end

endmodule
```

If you use the following Verilog HDL read statements instead of the `if-else` statements, the HDL code specifies that the read results in old data when a read operation and write operation occurs at the same time for the same address on the same port or mixed ports. This mode is supported only in device families that support M144, M9k, and MLAB memory blocks.

Example 12-19: VHDL Read Statement Example

```
always @ (posedge clk)
begin // Port A
  if (we_a)
     ram[addr_a] <= data_a;

  q_a <= ram[addr_a];
end

always @ (posedge clk)
begin // Port B
  if (we_b)
     ram[addr_b] <= data_b;

  q_b <= ram[addr_b];
end
```

The VHDL single-clock code sample i maps directly into Altera synchronous memory. When a read and write operation occurs on the same port for the same address, the new data being written to the memory is read. When a read and write operation occurs on different ports for the same address, the old data in the memory is read. Simultaneous write operations to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 12-20: VHDL True Dual-Port RAM with Single Clock (part 1)

```
    library ieee;
use ieee.std_logic_1164.all;

entity true_dual_port_ram_single_clock is
    generic (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 6
    );
    port (
        clk    : in std_logic;
        addr_a    : in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b    : in natural range 0 to 2**ADDR_WIDTH - 1;
        data_a    : in std_logic_vector((DATA_WIDTH-1) downto 0);
        data_b    : in std_logic_vector((DATA_WIDTH-1) downto 0);
        we_a    : in std_logic := '1';
        we_b    : in std_logic := '1';
        q_a    : out std_logic_vector((DATA_WIDTH -1) downto 0);
        q_b    : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );
end true_dual_port_ram_single_clock;

architecture rtl of true_dual_port_ram_single_clock is
    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
```

```
        type memory_t is array((2**ADDR_WIDTH - 1) downto 0) of word_t;
        -- Declare the RAM signal.
        shared variable ram : memory_t;
```

Example 12-21: VHDL True Dual-Port RAM with Single Clock (part 2)

```
        begin
        process(clk)
        begin
        if(rising_edge(clk)) then -- Port A
            if(we_a = '1') then
                ram(addr_a) <= data_a;

                -- Read-during-write on the same port returns NEW data
                q_a <= data_a;
            else
                -- Read-during-write on the mixed port returns OLD data
                q_a <= ram(addr_a);
            end if;
        end if;
        end process;

        process(clk)
        begin
        if(rising_edge(clk)) then -- Port B
            if(we_b = '1') then
                ram(addr_b) := data_b;
                -- Read-during-write on the same port returns NEW data
                q_b <= data_b;
            else
                -- Read-during-write on the mixed port returns OLD data
                q_b <= ram(addr_b);
            end if;
        end if;
        end process;

    end rtl;
```

Related Information

Check Read-During-Write Behavior on page 12-12

Mixed-Width Dual-Port RAM

The RAM code examples show SystemVerilog and VHDL code that infers RAM with data ports with different widths.

This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multi-dimensional array to model the different read width, write width, or both. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

The first dimension of the multi-dimensional packed array represents the ratio of the wider port to the narrower port, and the second dimension represents the narrower port width. The read and write port widths must specify a read or write ratio supported by the memory blocks in the target device, or the synthesis tool does not infer a RAM.

Refer to the Quartus II templates for parameterized examples that you can use for supported combinations of read and write widths, and true dual port RAM examples with two read ports and two write ports for mixed-width writes and reads.

Example 12-22: SystemVerilog Mixed-Width RAM with Read Width Smaller than Write Width

```
    module mixed_width_ram    // 256x32 write and 1024x8 read
(
        input [7:0] waddr,
        input [31:0] wdata,
        input we, clk,
        input [9:0] raddr,
        output [7:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
        begin
            if(we) ram[waddr] <= wdata;
            q <= ram[raddr / 4][raddr % 4];
        end
endmodule : mixed_width_ram
```

Example 12-23: SystemVerilog Mixed-Width RAM with Read Width Larger than Write Width

```
module mixed_width_ram     // 1024x8 write and 256x32 read
(
        input [9:0] waddr,
        input [31:0] wdata,
        input we, clk,
        input [7:0] raddr,
        output [9:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
         begin
            if(we) ram[waddr / 4][waddr % 4] <= wdata;
            q <= ram[raddr];
         end
endmodule : mixed_width_ram
```

Example 12-24: VHDL Mixed-Width RAM with Read Width Smaller than Write Width

```
library ieee;
use ieee.std_logic_1164.all;

package ram_types is
    type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
    type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;
```

```
entity mixed_width_ram is
    port (
        we, clk : in  std_logic;
        waddr   : in  integer range 0 to 255;
        wdata   : in  word_t;
        raddr   : in  integer range 0 to 1023;
        q       : out std_logic_vector(7 downto 0));
end mixed_width_ram;

architecture rtl of mixed_width_ram is
    signal ram : ram_t;
begin  -- rtl
    process(clk, we)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                ram(waddr) <= wdata;
            end if;
            q <= ram(raddr / 4 )(raddr mod 4);
        end if;
    end process;
end rtl;
```

Example 12-25: VHDL Mixed-Width RAM with Read Width Larger than Write Width

```
library ieee;
use ieee.std_logic_1164.all;

package ram_types is
    type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
    type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;

entity mixed_width_ram is
    port (
        we, clk : in  std_logic;
        waddr   : in  integer range 0 to 1023;
        wdata   : in  std_logic_vector(7 downto 0);
        raddr   : in  integer range 0 to 255;
        q       : out word_t);
end mixed_width_ram;

architecture rtl of mixed_width_ram is
    signal ram : ram_t;
begin  -- rtl
    process(clk, we)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                ram(waddr / 4)(waddr mod 4) <= wdata;
            end if;
            q <= ram(raddr);
        end if;
    end process;
end rtl;
```

RAM with Byte-Enable Signals

The RAM code examples show SystemVerilog and VHDL code that infers RAM with controls for writing single bytes into the memory word, or byte-enable signals.

Byte enables are modeled by creating write expressions with two indices and writing part of a RAM "word." With these implementations, you can also write more than one byte at once by enabling the appropriate byte enables.

This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multidimensional array. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

Refer to the Quartus II templates for parameterized examples that you can use for different address widths, and true dual port RAM examples with two read ports and two write ports.

Example 12-26: SystemVerilog Simple Dual-Port Synchronous RAM with Byte Enable

```
module byte_enabled_simple_dual_port_ram
(
    input we, clk,
    input [5:0] waddr, raddr, // address width = 6
    input [3:0] be,          // 4 bytes per word
    input [31:0] wdata,       // byte width = 8, 4 bytes per word
    output reg [31:0] q       // byte width = 8, 4 bytes per word
);
    // use a multi-dimensional packed array
    //to model individual bytes within the word
   logic [3:0][7:0] ram[0:63];    // # words = 1 << address width

   always_ff@(posedge clk)
   begin
       if(we) begin
           if(be[0]) ram[waddr][0] <= wdata[7:0];
           if(be[1]) ram[waddr][1] <= wdata[15:8];
           if(be[2]) ram[waddr][2] <= wdata[23:16];
       if(be[3]) ram[waddr][3] <= wdata[31:24];
       end
    q <= ram[raddr];
   end
endmodule
```

Example 12-27: VHDL Simple Dual-Port Synchronous RAM with Byte Enable

```
library ieee;
use ieee.std_logic_1164.all;
library work;

entity byte_enabled_simple_dual_port_ram is
port (
    we, clk : in  std_logic;
    waddr, raddr : in  integer range 0 to 63 ;     -- address width = 6
    be      : in  std_logic_vector (3 downto 0);   -- 4 bytes per word
    wdata   : in  std_logic_vector(31 downto 0);   -- byte width = 8
    q       : out std_logic_vector(31 downto 0) ); -- byte width = 8
end byte_enabled_simple_dual_port_ram;

architecture rtl of byte_enabled_simple_dual_port_ram is
  --  build up 2D array to hold the memory
```

```
    type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
    type ram_t is array (0 to 63) of word_t;

    signal ram : ram_t;
    signal q_local : word_t;

    begin  -- Re-organize the read data from the RAM to match the output
        unpack: for i in 0 to 3 generate
            q(8*(i+1) - 1 downto 8*i) <= q_local(i);
    end generate unpack;

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                if(be(0) = '1') then
                    ram(waddr)(0) <= wdata(7 downto 0);
                end if;
                if be(1) = '1' then
                    ram(waddr)(1) <= wdata(15 downto 8);
                end if;
                if be(2) = '1' then
                    ram(waddr)(2) <= wdata(23 downto 16);
                end if;
                if be(3) = '1' then
                    ram(waddr)(3) <= wdata(31 downto 24);
                end if;
            end if;
            q_local <= ram(raddr);
        end if;
    end process;
end rtl;
```

Specifying Initial Memory Contents at Power-Up

Your synthesis tool may offer various ways to specify the initial contents of an inferred memory.

There are slight power-up and initialization differences between dedicated RAM blocks and the MLAB memory due to the continuous read of the MLAB. Altera dedicated RAM block outputs always power-up to zero and are set to the initial value on the first read. For example, if address `0` is pre-initialized to FF, the RAM block powers up with the output at `0`. A subsequent read after power-up from address `0` outputs the pre-initialized value of FF. Therefore, if a RAM is powered up and an enable (read enable or clock enable) is held low, the power-up output of `0` is maintained until the first valid read cycle. The MLAB is implemented using registers that power-up to `0`, but are initialized to their initial value immediately at power-up or reset. Therefore, the initial value is seen, regardless of the enable status. The Quartus II software maps inferred memory to MLABs when the HDL code specifies an appropriate `ramstyle` attribute.

In Verilog HDL, you can use an initial block to initialize the contents of an inferred memory. Quartus II integrated synthesis automatically converts the initial block into a **.mif** file for the inferred RAM.

Example 12-28: Verilog HDL RAM with Initialized Contents

```
module ram_with_init(
   output reg [7:0] q,
   input [7:0] d,
   input [4:0] write_address, read_address,
   input we, clk
);
   reg [7:0] mem [0:31];
```

```
    integer i;

    initial begin
       for (i = 0; i < 32; i = i + 1)
          mem[i] = i[7:0];
    end

    always @ (posedge clk) begin
       if (we)
          mem[write_address] <= d;
       q <= mem[read_address];
    end
 endmodule
```

Quartus II integrated synthesis and other synthesis tools also support the `$readmemb` and `$readmemh` commands so that RAM initialization and ROM initialization work identically in synthesis and simulation.

Example 12-29: Verilog HDL RAM Initialized with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
    $readmemb("ram.txt", ram);
end
```

In VHDL, you can initialize the contents of an inferred memory by specifying a default value for the corresponding signal. Quartus II integrated synthesis automatically converts the default value into a .mif file for the inferred RAM.

Example 12-30: VHDL RAM with Initialized Contents

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

ENTITY ram_with_init IS
    PORT(
            clock: IN STD_LOGIC;
            data: IN UNSIGNED (7 DOWNTO 0);
            write_address: IN integer RANGE 0 to 31;
            read_address: IN integer RANGE 0 to 31;
            we: IN std_logic;
            q: OUT UNSIGNED (7 DOWNTO 0));
END;

ARCHITECTURE rtl OF ram_with_init IS

    TYPE MEM IS ARRAY(31 DOWNTO 0) OF unsigned(7 DOWNTO 0);
    FUNCTION initialize_ram
        return MEM is
        variable result : MEM;
    BEGIN
        FOR i IN 31 DOWNTO 0 LOOP
            result(i) := to_unsigned(natural(i), natural'(8));
        END LOOP;
        RETURN result;
    END initialize_ram;

    SIGNAL ram_block : MEM := initialize_ram;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
            ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
        END IF;
    END PROCESS;
END rtl;
```

Related Information

- Quartus II Integrated Synthesis on page 16-1

Inferring ROM Functions from HDL Code

ROMs are inferred when a `CASE` statement exists in which a value is set to a constant for every choice in the case statement.

Because small ROMs typically achieve the best performance when they are implemented using the registers in regular logic, each ROM function must meet a minimum size requirement to be inferred and placed into memory.

Note: If you use Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any ROM Size for Recognition** option in the **Advanced Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with synchronous memory blocks. For example, Quartus II integrated synthesis provides the `romstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.

Note: Because formal verification tools do not support ROM IP cores, Quartus II integrated synthesis does not infer ROM IP cores when a formal verification tool is selected. When you are using a formal verification flow, Altera recommends that you instantiate ROM IP core blocks in separate entities or modules that contain only the ROM logic, because you may need to treat the entity or module as a black box during formal verification. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; refer to the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Arria series, Cyclone series, or Stratix series devices and newer device families, either the address or the output must be registered for synthesis software to infer a ROM block. When your design uses output registers, the synthesis software implements registers from the input registers of the RAM block without affecting the functionality of the ROM. If you register the address, the power-up state of the inferred ROM can be different from the HDL design. In this scenario, the synthesis software issues a warning. The Quartus II Help explains the condition under which the functionality changes when you use Quartus II integrated synthesis.

The following ROM examples map directly to the Altera memory architecture.

Example 12-31: Verilog HDL Synchronous ROM

```
module sync_rom (clock, address, data_out);
    input clock;
    input [7:0] address;
    output [5:0] data_out;

    reg [5:0] data_out;

    always @ (posedge clock)
    begin
        case (address)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule
```

Example 12-32: VHDL Synchronous ROM

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

ENTITY sync_rom IS
    PORT (
        clock: IN STD_LOGIC;
        address: IN STD_LOGIC_VECTOR(7 downto 0);
        data_out: OUT STD_LOGIC_VECTOR(5 downto 0)
    );
END sync_rom;

ARCHITECTURE rtl OF sync_rom IS
BEGIN
PROCESS (clock)
    BEGIN
    IF rising_edge (clock) THEN
        CASE address IS
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS     => data_out <= "101111";
        END CASE;
    END IF;
    END PROCESS;
END rtl;
```

Example 12-33: Verilog HDL Dual-Port Synchronous ROM Using readmemb

```
module dual_port_rom (
    input [(addr_width-1):0] addr_a, addr_b,
    input clk,
    output reg [(data_width-1):0] q_a, q_b
);
    parameter data_width = 8;
    parameter addr_width = 8;

    reg [data_width-1:0] rom[2**addr_width-1:0];

    initial // Read the memory contents in the file
             //dual_port_rom_init.txt.
    begin
        $readmemb("dual_port_rom_init.txt", rom);
    end

    always @ (posedge clk)
    begin
        q_a <= rom[addr_a];
        q_b <= rom[addr_b];
    end
endmodule
```

Example 12-34: VHDL Dual-Port Synchronous ROM Using Initialization Function

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dual_port_rom is
    generic (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 8
    );
    port (
        clk        : in std_logic;
        addr_a    : in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b    : in natural range 0 to 2**ADDR_WIDTH - 1;
        q_a        : out std_logic_vector((DATA_WIDTH -1) downto 0);
        q_b        : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );
end entity;

architecture rtl of dual_port_rom is
    -- Build a 2-D array type for the ROM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is array(addr_a'high downto 0) of word_t;

    function init_rom
        return memory_t is
        variable tmp : memory_t := (others => (others => '0'));
    begin
        for addr_pos in 0 to 2**ADDR_WIDTH - 1 loop
            -- Initialize each address with the address itself
            tmp(addr_pos) := std_logic_vector(to_unsigned(addr_pos,
DATA_WIDTH));
        end loop;
        return tmp;
    end init_rom;

    -- Declare the ROM signal and specify a default initialization value.
    signal rom : memory_t := init_rom;
begin
    process(clk)
    begin
```

```
        if (rising_edge(clk)) then
            q_a <= rom(addr_a);
            q_b <= rom(addr_b);
        end if;
        end process;
    end rtl;
```

Related Information

- **Quartus II Integrated Synthesis** on page 16-1

Inferring Shift Registers in HDL Code

To infer shift registers, synthesis tools detect a group of shift registers of the same length and convert them to an Altera shift register IP core.

To be detected, all the shift registers must have the following characteristics:

- Use the same clock and clock enable
- Do not have any other secondary signals
- Have equally spaced taps that are at least three registers apart

When you use a formal verification flow, Altera recommends that you create shift register blocks in separate entities or modules containing only the shift register logic, because you might have to treat the entity or module as a black box during formal verification.

Note: Because formal verification tools do not support shift register IP cores, Quartus II integrated synthesis does not infer the Altera shift register IP core when a formal verification tool is selected. You can select EDA tools for use with your design on the **EDA Tool Settings** page of the **Settings** dialog box in the Quartus II software.

Synthesis recognizes shift registers only for device families that have dedicated RAM blocks, and the software uses certain guidelines to determine the best implementation.

Quartus II integrated synthesis uses the following guidelines which are common in other EDA tools. The Quartus II software determines whether to infer the Altera shift register IP core based on the width of the registered bus (`W`), the length between each tap (`L`), and the number of taps (`N`). If the **Auto Shift Register Recognition** setting is set to **Auto**, Quartus II integrated synthesis uses the **Optimization Technique** setting, logic and RAM utilization information about the design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are implemented in RAM blocks for logic.

- If the registered bus width is one (`W = 1`), the software infers shift register IP if the number of taps times the length between each tap is greater than or equal to 64 (`N x L > 64`).
- If the registered bus width is greater than one (`W > 1`), the software infers Altera shift register IP core if the registered bus width times the number of taps times the length between each tap is greater than or equal to 32 (`W × N × L > 32`).

If the length between each tap (`L`) is not a power of two, the software uses more logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or ALMs is required to implement the function. This decode logic eliminates the performance and utilization advantages of implementing shift registers in memory.

The registers that the software maps to the Altera shift register IP core and places in RAM are not available in a Verilog HDL or VHDL output file for simulation tools because their node names do not exist after synthesis.

Note: If your design uses a shift enable signal to infer a shift register, the shift register will not be implemented into MLAB memory, but can use only dedicated RAM blocks. You can use the `ramstyle` attribute to control the type of memory structure that implements the shift register.

Simple Shift Register

The code samples show a simple, single-bit wide, 64-bit long shift register.

The synthesis software implements the register (`W` = 1 and `M = 64`) in an ALTSHIFT_TAPS IP core for supported devices and maps it to RAM in supported devices, which may be placed in dedicated RAM blocks or MLAB memory. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 12-35: Verilog HDL Single-Bit Wide, 64-Bit Long Shift Register

```
module shift_1x64 (clk, shift, sr_in, sr_out);
    input clk, shift;
    input sr_in;
    output sr_out;

    reg [63:0] sr;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            sr[63:1] <= sr[62:0];
            sr[0] <= sr_in;
        end
    end
    assign sr_out = sr[63];
endmodule
```

Example 12-36: VHDL Single-Bit Wide, 64-Bit Long Shift Register

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY shift_1x64 IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC;
        sr_out: OUT STD_LOGIC
    );
END shift_1x64;

ARCHITECTURE arch OF shift_1x64 IS
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
    SIGNAL sr: sr_length;
BEGIN
    PROCESS (clk)
        BEGIN
        IF (clk'EVENT and clk = '1') THEN
            IF (shift = '1') THEN
            sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
            sr(0) <= sr_in;
            END IF;
        END IF;
    END PROCESS;
    sr_out <= sr(63);
END arch;
```

Shift Register with Evenly Spaced Taps

The following examples show a Verilog HDL and VHDL 8-bit wide, 64-bit long shift register (`W > 1` and `M = 64`) with evenly spaced taps at 15, 31, and 47.

The synthesis software implements this function in a single ALTSHIFT_TAPS IP core and maps it to RAM in supported devices, which is allowed placement in dedicated RAM blocks or MLAB memory.

Example 12-37: Verilog HDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
module shift_8x64_taps (clk, shift, sr_in, sr_out, sr_tap_one, sr_tap_two,
sr_tap_three );
    input clk, shift;
    input [7:0] sr_in;
    output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out;

    reg [7:0] sr [63:0];
    integer n;

     always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            for (n = 63; n>0; n = n-1)
            begin
                sr[n] <= sr[n-1];
            end
            sr[0] <= sr_in;
        end

    end
    assign sr_tap_one = sr[15];
    assign sr_tap_two = sr[31];
    assign sr_tap_three = sr[47];
    assign sr_out = sr[63];
endmodule
```

Example 12-38: VHDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY shift_8x64_taps IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_one: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_two : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_three: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS
    SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;
```

```
        SIGNAL sr: sr_length;
    BEGIN
        PROCESS (clk)
        BEGIN
            IF (clk'EVENT and clk = '1') THEN
                IF (shift = '1') THEN
                    sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
                    sr(0) <= sr_in;
                END IF;
            END IF;
        END PROCESS;
        sr_tap_one <= sr(15);
        sr_tap_two <= sr(31);
        sr_tap_three <= sr(47);
        sr_out <= sr(63);
    END arch;
```

Register and Latch Coding Guidelines

This section provides device-specific coding recommendations for Altera registers and latches.

Understanding the architecture of the target Altera device helps ensure that your code produces the expected results and achieves the optimal quality of results.

Register Power-Up Values in Altera Devices

Registers in the device core always power up to a low (`0`) logic level on all Altera devices.

If your design specifies a power-up level other than `0`, synthesis tools can implement logic that causes registers to behave as if they were powering up to a high (`1`) logic level.

If your design uses a preset signal on a device that does not support presets in the register architecture, your synthesis tool may convert the preset signal to a clear signal, which requires synthesis to perform an optimization referred to as NOT gate push-back. NOT gate push-back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high, and the device operates as expected. In this case, your synthesis tool may issue a message informing you about the power-up condition. The register itself powers up low, but the register output is inverted, so the signal that arrives at all destinations is high.

Due to these effects, if you specify a non-zero reset value, you may cause your synthesis tool to use the asynchronous clear (`aclr`) signals available on the registers to implement the high bits with NOT gate push-back. In that case, the registers look as though they power up to the specified reset value.

When an asynchronous load (`aload`) signal is available in the device registers, your synthesis tools can implement a reset of `1` or `0` value by using an asynchronous load of `1` or `0`. When the synthesis tool uses a load signal, it is not performing NOT gate push-back, so the registers power up to a `0` logic level.

For additional details, refer to the appropriate device family handbook or the appropriate handbook on the Altera website.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset. Altera recommends this practice to reset the device after power-up to restore the proper state.

You can make your design more stable and avoid potential glitches by synchronizing external or combinational logic of the device architecture before you drive the asynchronous control ports of registers.

Related Information

- Design Recommendations for Altera Devices and the Quartus II Design Assistant on page 11-1

Specifying a Power-Up Value

If you want to force a particular power-up condition for your design, you can use the synthesis options available in your synthesis tool.

With Quartus II integrated synthesis, you can apply the **Power-Up Level** logic option. You can also apply the option with an `altera_attribute` assignment in your source code. Using this option forces synthesis to perform NOT gate push-back because synthesis tools cannot actually change the power-up states of core registers.

You can apply the Quartus II integrated synthesis **Power-Up Level** logic option to a specific register or to a design entity, module, or subdesign. If you do so, every register in that block receives the value. Registers power up to `0` by default; therefore, you can use this assignment to force all registers to power up to `1` using NOT gate push-back.

Note: Setting the **Power-Up Level** to a logic level of high for a large design entity could degrade the quality of results due to the number of inverters that are required. In some situations, issues are caused by enable signal inference or secondary control logic inference. It may also be more difficult to migrate such a design to an ASIC.

Note: You can simulate the power-up behavior in a functional simulation if you use initialization.

Some synthesis tools can also read the default or initial values for registered signals and implement this behavior in the device. For example, Quartus II integrated synthesis converts default values for registered signals into **Power-Up Level** settings. When the Quartus II software reads the default values, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

Example 12-39: Verilog Register with High Power-Up Value

```
reg q = 1’b1; //q has a default value of ‘1’

always @ (posedge clk)
begin
    q <= d;
end
```

Example 12-40: VHDL Register with High Power-Up Level

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
    IF (rising_edge(clk)) THEN
        q <= d;
    END IF;
END PROCESS;
```

There may also be undeclared default power-up conditions based on signal type. If you declare a VHDL register signal as an integer, Quartus II synthesis attempts to use the left end of the integer range as the power-up value. For the default signed integer type, the default power-up value is the highest magnitude negative integer (`100…001`). For an unsigned integer type, the default power-up value is `0`.

Note: If the target device architecture does not support two asynchronous control signals, such as `aclr` and `aload`, you cannot set a different power-up state and reset state. If the NOT gate push-back algorithm creates logic to set a register to `1`, that register will power-up high. If you set a different power-up condition through a synthesis assignment or initial value, the power-up level is ignored during synthesis.

Related Information

- **Quartus II Integrated Synthesis** on page 16-1

Secondary Register Control Signals Such as Clear and Clock Enable

The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells.

The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells. Device families vary in their support for secondary signals, so consult the device family data sheet to verify which signals are available in your target device.

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. The control signals have a certain priority due to the nature of the architecture, so your HDL code should follow that priority where possible.

Your synthesis tool can emulate any control signals using regular logic, so achieving functionally correct results is always possible. However, if your design requirements are flexible in terms of which control signals are used and in what priority, match your design to the target device architecture to achieve the most efficient results. If the priority of the signals in your design is not the same as that of the target architecture, extra logic may be required to implement the control signals. This extra logic uses additional device resources and can cause additional delays for the control signals.

In addition, there are certain cases where using logic other than the dedicated control logic in the device architecture can have a larger impact. For example, the clock enable signal has priority over the synchronous reset or clear signal in the device architecture. The clock enable turns off the clock line in the LAB, and the clear signal is synchronous. Therefore, in the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a synchronous clear signal that has priority over the clock enable signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the clock enable port of a register, you cannot apply a Clock Enable Multicycle constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals, and using a different priority causes unexpected results with an assignment to the clock enable signal.

Note: The priority order for secondary control signals in Altera devices differs from the order for other vendors' devices. If your design requirements are flexible regarding priority, verify that the secondary control signals meet design performance requirements when migrating designs between FPGA vendors and try to match your target device architecture to achieve the best results.

The signal order is the same for all Altera device families, although, as noted previously, not all device families provide every signal. The following priority order is observed:

1. Asynchronous Clear, `aclr`—highest priority
2. Asynchronous Load, `aload`
3. Enable, `ena`
4. Synchronous Clear, `sclr`
5. Synchronous Load, `sload`
6. Data In, `data`—lowest priority

The following examples provide Verilog HDL and VHDL code that creates a register with the `aclr`, `aload`, and `ena` control signals.

Note: The Verilog HDL example does not have `adata` on the sensitivity list, but the VHDL example does. This is a limitation of the Verilog HDL language—there is no way to describe an asynchronous load signal (in which `q` toggles if `adata` toggles while `aload` is high). All synthesis tools should infer an `aload` signal from this construct despite this limitation. When they perform such inference, you may see information or warning messages from the synthesis tool.

Example 12-41: Verilog HDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals

```
module dff_control(clk, aclr, aload, ena, data, adata, q);
    input clk, aclr, aload, ena, data, adata;
    output q;

    reg q;

    always @ (posedge clk or posedge aclr or posedge aload)
    begin
        if (aclr)
            q <= 1'b0;
        else if (aload)
            q <= adata;
        else if (ena)
            q <= data;
    end
endmodule
```

Example 12-42: VHDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals (part 1)

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

ENTITY dff_control IS
    PORT (
        clk: IN STD_LOGIC;
        aclr: IN STD_LOGIC;
        aload: IN STD_LOGIC;
        adata: IN STD_LOGIC;
        ena: IN STD_LOGIC;
      data: IN STD_LOGIC;
q: OUT STD_LOGIC
```

```
    );
END dff_control;
```

Example 12-43: VHDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals (part 2)

```
ARCHITECTURE rtl OF dff_control IS
BEGIN
    PROCESS (clk, aclr, aload, adata)
    BEGIN
IF (aclr = '1') THEN
q <= '0';
ELSIF (aload = '1') THEN
q <= adata;
ELSE
            IF (clk = '1' AND clk'event) THEN
                IF (ena     ='1') THEN
q <= data;
                END IF;
            END IF;
        END IF;
    END PROCESS;
END rtl;
```

Latches

A latch is a small combinational loop that holds the value of a signal until a new value is assigned.

Latches can be inferred from HDL code when you did not intend to use a latch. If you do intend to infer a latch, it is important to infer it correctly to guarantee correct device operation.

Note: Altera recommends that you design without the use of latches whenever possible.

Related Information

- **Recommended Design Practices** on page 11-1

Avoid Unintentional Latch Generation

When you are designing combinational logic, certain coding styles can create an unintentional latch.

For example, when `CASE` or `IF` statements do not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. Check your synthesis tool messages for references to inferred latches. If your code unintentionally creates a latch, make code changes to remove the latch.

A latch is required if a signal is assigned a value outside of a clock edge (for example, with an asynchronous reset), but is not assigned a value in an edge-triggered design block. An unintentional latch may be generated if your HDL code assigns a value to a signal in an edge-triggered design block, but that logic is removed during synthesis. For example, when a `CASE` or `IF` statement tests the value of a condition with a parameter or generic that evaluates to FALSE, any logic or signal assignment in that statement is not required and is optimized away during synthesis. This optimization may result in a latch being generated for the signal.

Note: Latches have limited support in formal verification tools. Therefore, ensure that you do not infer latches unintentionally.

The `full_case` attribute can be used in Verilog HDL designs to treat unspecified cases as don't care values (`X`). However, using the `full_case` attribute can cause simulation mismatches because this attribute is a synthesis-only attribute, so simulation tools still treat the unspecified cases as latches.

Omitting the final `else` or `when others` clause in an `if` or `case` statement can also generate a latch. Don't care (`X`) assignments on the default conditions are useful in preventing latch generation. For the best logic optimization, assign the default `case` or final `else` value to don't care (`X`) instead of a logic value.

Without the final `else` clause, the following code creates unintentional latches to cover the remaining combinations of the `sel` inputs. When you are targeting a Stratix device with this code, omitting the final `else` condition can cause the synthesis software to use up to six LEs, instead of the three it uses with the `else` statement. Additionally, assigning the final `else` clause to `1` instead of `X` can result in slightly more LEs, because the synthesis software cannot perform as much optimization when you specify a constant value compared to a don't care value.

Example 12-44: VHDL Code Preventing Unintentional Latch Creation

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

ENTITY nolatch IS
    PORT (a,b,c: IN STD_LOGIC;
        sel: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        oput: OUT STD_LOGIC);
END nolatch;

ARCHITECTURE rtl OF nolatch IS
BEGIN
    PROCESS (a,b,c,sel) BEGIN
        if sel = "00000" THEN
            oput <= a;
        ELSIF sel = "00001" THEN
            oput <= b;
        ELSIF sel = "00010" THEN
            oput <= c;
        ELSE                   --- Prevents latch inference
            oput <= ''X'; --/
        END if;
    END PROCESS;
END rtl;
```

Related Information

- Quartus II Integrated Synthesis on page 16-1

Inferring Latches Correctly

Synthesis tools can infer a latch that does not exhibit the glitch and timing hazard problems typically associated with combinational loops. When using Quartus II integrated synthesis, latches that are inferred by the software are reported in the **User-Specified and Inferred Latches** section of the Compilation Report. This report indicates whether the latch is considered safe and free of timing hazards.

Note: Timing analysis does not completely model latch timing in some cases. Do not use latches unless required by your design, and you fully understand the impact of using the latches.

If a latch or combinational loop in your design is not listed in the **User Specified and Inferred Latches** section, it means that it was not inferred as a safe latch by the software and is not considered glitch-free.

All combinational loops listed in the **Analysis & Synthesis Logic Cells Representing Combinational Loops** table in the Compilation Report are at risk of timing hazards. These entries indicate possible problems with your design that you should investigate. However, it is possible to have a correct design that includes combinational loops. For example, it is possible that the combinational loop cannot be sensitized. This can occur in cases where there is an electrical path in the hardware, but either the designer knows that the circuit never encounters data that causes that path to be activated, or the surrounding logic is set up in a mutually exclusive manner that prevents that path from ever being sensitized, independent of the data input.

For macrocell-based devices, all data (D-type) latches and set-reset (S-R) latches listed in the **Analysis & Synthesis User Specified and Inferred Latches** table have an implementation free of timing hazards, such as glitches. The implementation includes both a cover term to ensure there is no glitching and a single macrocell in the feedback loop.

For 4-input LUT-based devices, such as Stratixdevices, the Cyclone series, and MAX II devices, all latches in the **User Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value, such as a D-type input toggling when the enable input is inactive or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is required for a loop around a single LUT. The Quartus II software uses a single LUT in the feedback loop whenever possible. A latch that has data, enable, set, and reset inputs in addition to the output fed back to the input cannot be implemented in a single 4-input LUT. If the Quartus II software cannot implement the latch with a single-LUT loop because there are too many inputs, the **User Specified and Inferred Latches** table indicates that the latch is not free of timing hazards.

For 6-input LUT-based devices, the software can implement all latch inputs with a single adaptive look-up table (ALUT) in the combinational loop. Therefore, all latches in the **User-Specified and Inferred Latches** table are free of timing hazards when a single input changes.

If a latch is listed as a safe latch, other optimizations performed by the Quartus II software, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance. To ensure hazard-free behavior, only one control input can change at a time. Changing two inputs simultaneously, such as deasserting set and reset at the same time, or changing data and enable at the same time, can produce incorrect behavior in any latch.

Quartus II integrated synthesis infers latches from `always` blocks in Verilog HDL and `process` statements in VHDL, but not from continuous assignments in Verilog HDL or concurrent signal assignments in VHDL. These rules are the same as for register inference. The software infers registers or flipflops only from `always` blocks and `process` statements.

Example 12-45: Verilog HDL Set-Reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1
      else if (ResetTerm)
         LatchOut = 1'b0
```

```
    end
endmodule
```

Example 12-46: VHDL Data Type Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY simple_latch IS
    PORT (
        enable, data    : IN STD_LOGIC;
        q               : OUT STD_LOGIC
    );
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

    latch : PROCESS (enable, data)
        BEGIN
        IF (enable = '1') THEN
            q <= data;
        END IF;
    END PROCESS latch;
END rtl;
```

The following example shows a Verilog HDL continuous assignment that does not infer a latch in the Quartus II software:

Example 12-47: VHDL Continuous Assignment Does Not Infer Latch

```
assign latch_out = (~en & latch_out) | (en & data);
```

The behavior of the assignment is similar to a latch, but it may not function correctly as a latch, and its timing is not analyzed as a latch. Quartus II integrated synthesis also creates safe latches when possible for instantiations of an Altera latch IP core. You can use an Altera latch IP core to define a latch with any combination of data, enable, set, and reset inputs. The same limitations apply for creating safe latches as for inferring latches from HDL code.

Inferring Altera latch IP core in another synthesis tool ensures that the implementation is also recognized as a latch in the Quartus II software. If a third-party synthesis tool implements a latch using the Altera latch IP core, the Quartus II integrated synthesis lists the latch in the **User-Specified and Inferred Latches** table in the same way as it lists latches created in HDL source code. The coding style necessary to produce an Altera latch IP core implementation may depend on your synthesis tool. Some third-party synthesis tools list the number of Altera latch IP cores that are inferred.

For LUT-based families, the Fitter uses global routing for control signals, including signals that Analysis and Synthesis identifies as latch enables. In some cases the global insertion delay may decrease the timing performance. If necessary, you can turn off the **Quartus II Global Signal** logic option to manually prevent the use of global signals. Global latch enables are listed in the **Global & Other Fast Signals** table in the Compilation Report.

General Coding Guidelines

This section describes how coding styles impacts synthesis of HDL code into the target Altera device.

Following Altera recommended coding styles, and in some cases designing logic structures to match the appropriate device architecture, can provide significant improvements in your design's efficiency and performance.

Tri-State Signals

When you target Altera devices, you should use tri-state signals only when they are attached to top-level bidirectional or output pins.

Avoid lower-level bidirectional pins, and avoid using the `Z` logic value unless it is driving an output or bidirectional pin. Synthesis tools implement designs with internal tri-state signals correctly in Altera devices using multiplexer logic, but Altera does not recommend this coding practice.

Note: In hierarchical block-based or incremental design flows, a hierarchical boundary cannot contain any bidirectional ports, unless the lower-level bidirectional port is connected directly through the hierarchy to a top-level output pin without connecting to any other design logic. If you use boundary tri-states in a lower-level block, synthesis software must push the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower-level tri-states are restricted with block-based design methodologies.

Clock Multiplexing

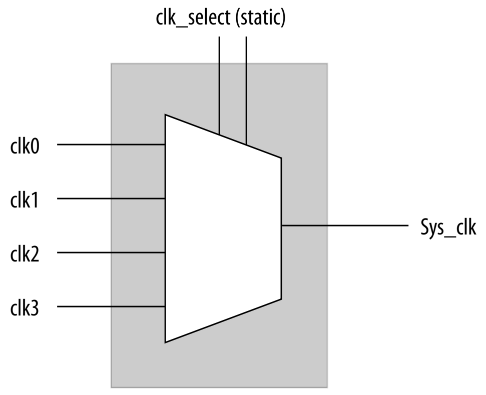
Clock multiplexing is sometimes used to operate the same logic function with different clock sources.

This type of logic can introduce glitches that create functional problems, and the delay inherent in the combinational logic can lead to timing problems. Clock multiplexers trigger warnings from a wide range of design rule check and timing analysis tools.

Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or the Clock Control Block available in certain Altera devices. These dedicated hardware blocks avoid glitches, ensure that you use global low-skew routing lines, and avoid any possible hold time problems on the device due to logic delay on the clock line. Many Altera devices also support dynamic PLL reconfiguration, which is the safest and most robust method of changing clock rates during device operation.

If you implement a clock multiplexer in logic cells because the design has too many clocks to use the clock control block, or if dynamic reconfiguration is too complex for your design, it is important to consider simultaneous toggling inputs and ensure glitch-free transitions.

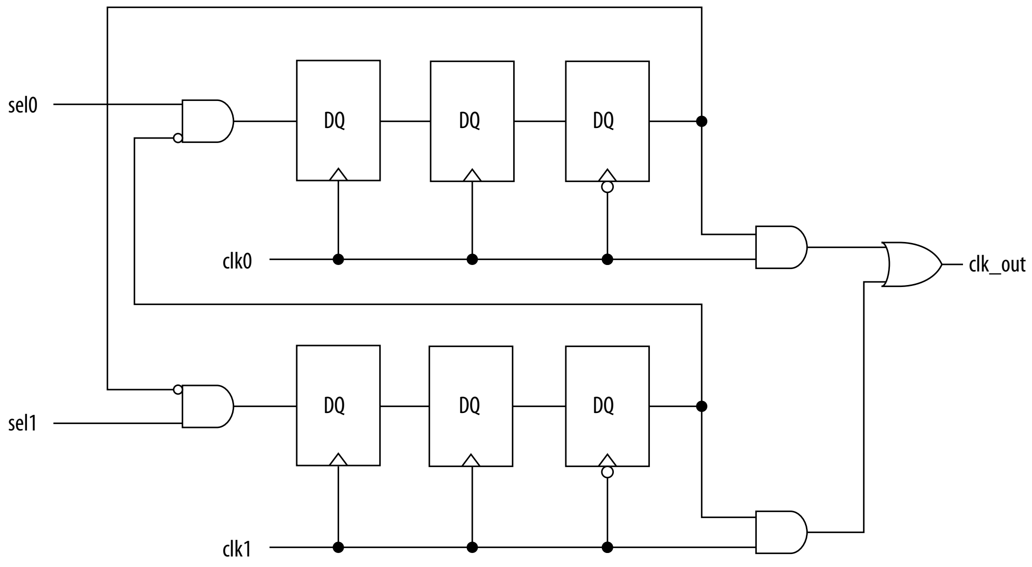
Figure 12-2: Simple Clock Multiplexer in a 6-Input LUT



The data sheet for your target device describes how LUT outputs may glitch during a simultaneous toggle of input signals, independent of the LUT function. Although, in practice, the 4:1 MUX function does not generate detectable glitches during simultaneous data input toggles, it is possible to construct cell implementations that do exhibit significant glitches, so this simple clock mux structure is not recommended. An additional problem with this implementation is that the output behaves erratically during a change in the `clk_select` signals. This behavior could create timing violations on all registers fed by the system clock and result in possible metastability.

A more sophisticated clock select structure can eliminate the simultaneous toggle and switching problems.

Figure 12-3: Glitch-Free Clock Multiplexer Structure



You can generalize this structure for any number of clock channels. The design ensures that no clock activates until all others are inactive for at least a few cycles, and that activation occurs while the clock is low. The design applies a `synthesis_keep` directive to the AND gates on the right side, which ensures there are no simultaneous toggles on the input of the `clk_out` OR gate.

Note: Switching from clock `A` to clock `B` requires that clock `A` continue to operate for at least a few cycles. If the old clock stops immediately, the design sticks. The select signals are implemented as a "one-hot" control in this example, but you can use other encoding if you prefer. The input side logic is asynchronous and is not critical. This design can tolerate extreme glitching during the switch process.

Example 12-48: Verilog HDL Clock Multiplexing Design to Avoid Glitches

```
module clock_mux (clk,clk_select,clk_out);

    parameter num_clocks = 4;

    input [num_clocks-1:0] clk;
    input [num_clocks-1:0] clk_select; // one hot
    output clk_out;

    genvar i;

    reg [num_clocks-1:0] ena_r0;
    reg [num_clocks-1:0] ena_r1;
    reg [num_clocks-1:0] ena_r2;
    wire [num_clocks-1:0] qualified_sel;

    // A look-up-table (LUT) can glitch when multiple inputs
    // change simultaneously. Use the keep attribute to
    // insert a hard logic cell buffer and prevent
    // the unrelated clocks from appearing on the same LUT.

    wire [num_clocks-1:0] gated_clks /* synthesis keep */;

    initial begin
        ena_r0 = 0;
        ena_r1 = 0;
        ena_r2 = 0;
    end

    generate
        for (i=0; i<num_clocks; i=i+1)
        begin : lp0
            wire [num_clocks-1:0] tmp_mask;
            assign tmp_mask = {num_clocks{1'b1}} ^ (1 << i);

            assign qualified_sel[i] = clk_select[i] & (~|(ena_r2 &
tmp_mask));

            always @(posedge clk[i]) begin
                ena_r0[i] <= qualified_sel[i];
                ena_r1[i] <= ena_r0[i];
            end

            always @(negedge clk[i]) begin
                ena_r2[i] <= ena_r1[i];
            end

            assign gated_clks[i] = clk[i] & ena_r2[i];
        end
    endgenerate

    // These will not exhibit simultaneous toggle by construction
    assign clk_out = |gated_clks;

endmodule
```

Related Information

Altera IP Core Literature

Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can provide significant improvements in your design's efficiency and performance.

A good example of an application using a large adder tree is a finite impulse response (FIR) correlator. Using a pipelined binary or ternary adder tree appropriately can greatly improve the quality of your results.

This section explains why coding recommendations are different for Altera 4-input LUT devices and 6-input LUT devices.

Architectures with 4-Input LUTs in Logic Elements

Architectures such as Stratix devices and the Cyclone series of devices contain 4-input LUTs as the standard combinational structure in the LE.

If your design can tolerate pipelining, the fastest way to add three numbers `A`, `B`, and `C` in devices that use 4-input lookup tables is to add`A + B`, register the output, and then add the registered output to `C`. Adding `A + B` takes one level of logic (one bit is added in one LE), so this runs at full clock speed. This can be extended to as many numbers as desired.

Adding five numbers in devices that use 4-input lookup tables requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Architectures with 6-Input LUTs in Adaptive Logic Modules

High-performance Altera device families use a 6-input LUT in their basic logic structure. These devices benefit from a different coding style from the previous example presented for 4-input LUTs.

Specifically, in these devices, ALMs can simultaneously add three bits. Therefore, the tree must be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Although the code in the previous example compiles successfully for 6-input LUT devices, the code is inefficient and does not take advantage of the 6-input adaptive ALUT. By restructuring the tree as a ternary tree, the design becomes much more efficient, significantly improving density utilization. Therefore, when you are targeting with ALUTs and ALMs, large pipelined binary adder trees designed for 4-input LUT architectures should be rewritten to take advantage of the advanced device architecture.

Note: You cannot pack a LAB full when using this type of coding style because of the number of LAB inputs. However, in a typical design, the Quartus II Fitter can pack other logic into each LAB to take advantage of the unused ALMs.

These examples show pipelined adders, but partitioning your addition operations can help you achieve better results in nonpipelined adders as well. If your design is not pipelined, a ternary tree provides much better performance than a binary tree. For example, depending on your synthesis tool, the HDL code `sum = (A + B + C) + (D + E)` is more likely to create the optimal implementation of a 3-input adder for `A + B + C` followed by a 3-input adder for `sum1 + D + E` than the code without the parentheses. If you do not add the parentheses, the synthesis tool may partition the addition in a way that is not optimal for the architecture.

Example 12-49: Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
```

```
    input clk, reset;
    input [3:0] in_1, in_2;
    output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
            state <= next_state;
    end
    always @ (*)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
            state_0: begin
               tmp_out_2 = in_1 + 5'b00001;
               next_state = state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state = state_2;
                    tmp_out_2 = tmp_out_0;
                end
                else begin
                    next_state = state_3;
                    tmp_out_2 = tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 = tmp_out_0 - 5'b00001;
                next_state = state_3;
            end
            state_3: begin
                tmp_out_2 = tmp_out_1 + 5'b00001;
                next_state = state_0;
            end
            state_4:begin
                tmp_out_2 = in_2 + 5'b00001;
                next_state = state_0;
            end
            default:begin
                tmp_out_2 = 5'b00000;
                next_state = state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `‘define` instead of the `parameter` data type, as follows:

```
‘define state_0 3'b000
‘define state_1 3'b001
‘define state_2 3'b010
```

```
‘define state_3 3'b011
‘define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a ‘`state_x` instead of a `state_x`, for example:

```
next_state <= ‘state_3;
```

Note: Although the `‘define` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

State Machine HDL Guidelines

Synthesis tools can recognize and encode Verilog HDL and VHDL state machines during synthesis. This section presents guidelines to ensure the best results when you use state machines.

Ensuring that your synthesis tool recognizes a piece of code as a state machine allows the tool to recode the state variables to improve the quality of results, and allows the tool to use the known properties of state machines to optimize other parts of the design. When synthesis recognizes a state machine, it is often able to improve the design area and performance.

To achieve the best results on average, synthesis tools often use one-hot encoding for FPGA devices and minimal-bit encoding for CPLD devices, although the choice of implementation can vary for different state machines and different devices. Refer to your synthesis tool documentation for specific ways to control the manner in which state machines are encoded.

To ensure proper recognition and inference of state machines and to improve the quality of results, Altera recommends that you observe the following guidelines, which apply to both Verilog HDL and VHDL:

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- If your design contains an operation that is used by more than one state, define the operation outside the state machine and cause the output logic of the state machine to use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus II software generates regular logic rather than inferring a state machine.

If a state machine enters an illegal state due to a problem with the device, the design likely ceases to function correctly until the next reset of the state machine. Synthesis tools do not provide for this situation by default. The same issue applies to any other registers if there is some kind of fault in the system. A `default` or `when others` clause does not affect this operation, assuming that your design never deliberately enters this state. Synthesis tools remove any logic generated by a default state if it is not reachable by normal state machine operation.

Many synthesis tools (including Quartus II integrated synthesis) have an option to implement a safe state machine. The software inserts extra logic to detect an illegal state and force the state machine’s transition to the reset state. It is commonly used when the state machine can enter an illegal state. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a dual-clock FIFO.

This option protects only state machines by forcing them into the reset state. All other registers in the design are not protected this way. If the design has asynchronous inputs, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

Related Information

- Quartus II Integrated Synthesis on page 16-1

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog HDL guidelines.

Some of these guidelines may be specific to Quartus II integrated synthesis. Refer to your synthesis tool documentation for specific coding recommendations. If the state machine is not recognized and inferred by the synthesis software (such as Quartus II integrated synthesis), the state machine is implemented as regular logic gates and registers, and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

- If you are using the SystemVerilog standard, use enumerated types to describe state machines.
- Represent the states in a state machine with the parameter data types in Verilog-1995 and Verilog-2001, and use the parameters to make state assignments. This parameter implementation makes the state machine easier to read and reduces the risk of errors during coding.
- Altera recommends against the direct use of integer values for state variables, such as `next_state <= 0`. However, using an integer does not prevent inference in the Quartus II software.
- No state machine is inferred in the Quartus II software if the state transition logic uses arithmetic similar to that in the following example:

```
case (state)
    0: begin
        if (ena) next_state <= state + 2;
        else next_state <= state + 1;
    end
    1: begin
    ...
endcase
```

case (state)0: beginif (ena) next_state <= state + 2;else next_state <= state + 1;end1: begin...endcase

- No state machine is inferred in the Quartus II software if the state variable is an output.
- No state machine is inferred in the Quartus II software for signed variables.

Related Information

- Verilog-2001 State Machine Coding Example on page 12-47
- SystemVerilog State Machine Coding Example on page 12-49

Verilog-2001 State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation.

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in_1` and `in_2` is an output of the state machine in `state_1` and `state_2`. The difference (`in_1` – `in_2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in_1` and `in_2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 12-50: Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk, reset;
    input [3:0] in_1, in_2;
    output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
            state <= next_state;
    end
    always @ (*)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
            state_0: begin
               tmp_out_2 = in_1 + 5'b00001;
               next_state = state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state = state_2;
                    tmp_out_2 = tmp_out_0;
                end
                else begin
                    next_state = state_3;
                    tmp_out_2 = tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 = tmp_out_0 - 5'b00001;
                next_state = state_3;
            end
            state_3: begin
                tmp_out_2 = tmp_out_1 + 5'b00001;
                next_state = state_0;
            end
            state_4:begin
                tmp_out_2 = in_2 + 5'b00001;
                next_state = state_0;
            end
            default:begin
                tmp_out_2 = 5'b00000;
                next_state = state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `‘define` instead of the `parameter` data type, as follows:

```
‘define state_0 3'b000
‘define state_1 3'b001
‘define state_2 3'b010
‘define state_3 3'b011
‘define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a `‘state_x` instead of a `state_x`, for example:

```
next_state <= ‘state_3;
```

Note: Although the `‘define` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

SystemVerilog State Machine Coding Example

The module `enum_fsm` is an example of a SystemVerilog state machine implementation that uses enumerated types. Altera recommends using this coding style to describe state machines in SystemVerilog.

Note: In Quartus II integrated synthesis, the enumerated type that defines the states for the state machine must be of an unsigned integer type. If you do not specify the enumerated type as `int unsigned`, a signed `int` type is used by default. In this case, the Quartus II integrated synthesis synthesizes the design, but does not infer or optimize the logic as a state machine.

Example 12-51: SystemVerilog State Machine Using Enumerated Types

```
module enum_fsm (input clk, reset, input int data[3:0], output int o);

enum int unsigned { S0 = 0, S1 = 2, S2 = 4, S3 = 8 } state, next_state;

always_comb begin : next_state_logic
      next_state = S0;
      case(state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S3;
      endcase
end

always_comb begin
      case(state)
         S0: o = data[3];
         S1: o = data[2];
         S2: o = data[1];
         S3: o = data[0];
      endcase
end
```

```
always_ff@(posedge clk or negedge reset) begin
      if(~reset)
         state <= S0;
      else
         state <= next_state;
end
endmodule
```

VHDL State Machines

To ensure proper recognition and inference of VHDL state machines, represent the states in a state machine with enumerated types and use the corresponding types to make state assignments.

This implementation makes the state machine easier to read and reduces the risk of errors during coding. If the state is not represented by an enumerated type, synthesis software (such as Quartus II integrated synthesis) does not recognize the state machine. Instead, the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

VHDL State Machine Coding Example

The following state machine has five states. The asynchronous reset sets the variable `state` to `state_0`.

The sum of `in1` and `in2` is an output of the state machine in `state_1` and `state_2`. The difference (`in1 - in2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in1` and `in2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 12-52: VHDL State Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY vhdl_fsm IS
   PORT(
      clk: IN STD_LOGIC;
      reset: IN STD_LOGIC;
      in1: IN UNSIGNED(4 downto 0);
      in2: IN UNSIGNED(4 downto 0);
      out_1: OUT UNSIGNED(4 downto 0)
      );
END vhdl_fsm;
ARCHITECTURE rtl OF vhdl_fsm IS
   TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);
   SIGNAL state: Tstate;
   SIGNAL next_state: Tstate;
BEGIN
   PROCESS(clk, reset)
   BEGIN
      IF reset = '1' THEN
            state <=state_0;
      ELSIF rising_edge(clk) THEN
            state <= next_state;
      END IF;
   END PROCESS;
PROCESS (state, in1, in2)
      VARIABLE tmp_out_0: UNSIGNED (4 downto 0);
      VARIABLE tmp_out_1: UNSIGNED (4 downto 0);
   BEGIN
```

```
            tmp_out_0 := in1 + in2;
            tmp_out_1 := in1 - in2;
            CASE state IS
               WHEN state_0 =>
                  out_1 <= in1;
                  next_state <= state_1;
               WHEN state_1 =>
                  IF (in1 < in2) then
                     next_state <= state_2;
                     out_1 <= tmp_out_0;
                  ELSE
                     next_state <= state_3;
                     out_1 <= tmp_out_1;
                  END IF;
               WHEN state_2 =>
                  IF (in1 < "0100") then
                     out_1 <= tmp_out_0;
                  ELSE
                     out_1 <= tmp_out_1;
                  END IF;
                     next_state <= state_3;
               WHEN state_3 =>
                     out_1 <= "11111";
                     next_state <= state_4;
               WHEN state_4 =>
                     out_1 <= in2;
                     next_state <= state_0;
               WHEN OTHERS =>
                     out_1 <= "00000";
                     next_state <= state_0;
            END CASE;
         END PROCESS;
      END rtl;
```

Multiplexer HDL Guidelines

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexer logic, you ensure the most efficient implementation in your Altera device.

This section addresses common problems and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes various types of multiplexers, and how they are implemented.

For more information, refer to the *Advanced Synthesis Cookbook*.

Related Information

Advanced Synthesis Cookbook

Quartus II Software Option for Multiplexer Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that extracts and optimizes buses of multiplexers during synthesis.

The default setting **Auto** for this option uses the optimization when it is most likely to benefit the optimization targets for your design. You can turn the option on or off specifically to have more control over its use.

Even with this Quartus II-specific option turned on, it is beneficial to understand how your coding style can be interpreted by your synthesis tool, and avoid the situations that can cause problems in your design.

Related Information

- **Quartus II Integrated Synthesis** on page 16-1

Multiplexer Types

This section addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs.

These HDL structures create different types of multiplexers, including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code, and how they might be implemented during synthesis, is the first step toward optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits.

Stratix series devices starting with the Stratix II device family feature 6-input look up tables (LUTs) which are perfectly suited for 4:1 multiplexer building blocks (4 data and 2 select inputs). The extended input mode facilitates implementing 8:1 blocks, and the fractured mode handles residual 2:1 multiplexer pairs. For device families using 4-input LUTs, such as the Cyclone series and Stratix devices, the 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers are decomposed by the synthesis tool into 4:1 multiplexer blocks, possibly with a residual 2:1 multiplexer at the head.

Example 12-53: Verilog HDL Binary-Encoded Multiplexers

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

Selector Multiplexers

Selector multiplexers have a separate select line for each data input.

The select lines for the multiplexer are one-hot encoded. Selector multiplexers are commonly built as a tree of AND and OR gates. An N-input selector multiplexer of this structure is slightly less efficient in implementation than a binary multiplexer. However, in many cases the select signal is the output of a decoder, in which case Quartus II Synthesis will try to combine the selector and decoder into a binary multiplexer.

Example 12-54: Verilog HDL One-Hot-Encoded Case Statement

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

Priority Multiplexers

In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority.

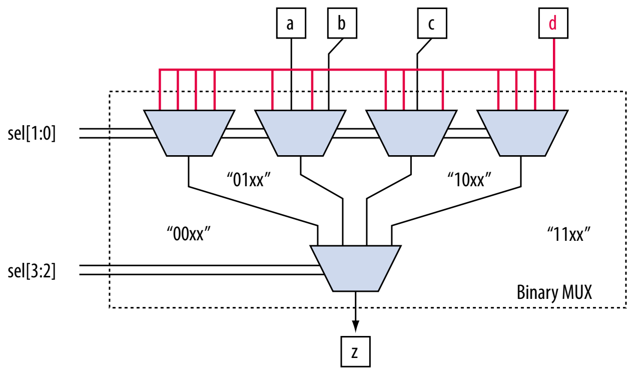
These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL.

Example 12-55: VHDL IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

The multiplexers form a chain, evaluating each condition or select bit sequentially.

Figure 12-4: Priority Multiplexer Implementation of an IF Statement



Depending on the number of multiplexers in the chain, the timing delay through this chain can become large, especially for device families with 4-input LUTs.

To improve the timing delay through the multiplexer, avoid priority multiplexers if priority is not required. If the order of the choices is not important to the design, use a `CASE` statement to implement a binary or selector multiplexer instead of a priority multiplexer. If delay through the structure is important in a multiplexed design requiring priority, consider recoding the design to reduce the number of logic levels to minimize delay, especially along your critical paths.

Implicit Defaults in If Statements

The `IF` statements in Verilog HDL and VHDL can be a convenient way to specify conditions that do not easily lend themselves to a `CASE`-type approach.

However, using `IF` statements can result in complicated multiplexer trees that are not easy for synthesis tools to optimize. In particular, every `IF` statement has an implicit `ELSE` condition, even when it is not specified. These implicit defaults can cause additional complexity in a multiplexed design.

There are several ways you can simplify multiplexed logic and remove unneeded defaults. The optimal method may be to recode the design so the logic takes the structure of a 4:1 `CASE` statement. Alternatively, if priority is important, you can restructure the code to reduce default cases and flatten the multiplexer. Examine whether the default "`ELSE IF`" conditions are don't care cases. You may be able to create a default `ELSE` statement to make the behavior explicit. Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

Default or Others Case Assignment

To fully specify the cases in a `CASE` statement, include a `default` (Verilog HDL) or `OTHERS` (VHDL) assignment.

This assignment is especially important in one-hot encoding schemes where many combinations of the select lines are unused. Specifying a case for the unused select line combinations gives the synthesis tool information about how to synthesize these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not require that the outcome in the unused cases be considered, often because designers assume these cases will not occur. For these types of designs, you can specify any value for the `default` or `OTHERS` assignment. However, be aware that the assignment value you choose can have a large effect on the logic utilization required to implement the design due to the different ways synthesis tools treat different values for the assignment, and how the synthesis tools use different speed and area optimizations.

To obtain best results, explicitly define invalid `CASE` selections with a separate `default` or `OTHERS` statement instead of combining the invalid cases with one of the defined cases.

If the value in the invalid cases is not important, specify those cases explicitly by assigning the `X` (don't care) logic value instead of choosing another value. This assignment allows your synthesis tool to perform the best area optimizations.

Cyclic Redundancy Check Functions

CRC computations are used heavily by communications protocols and storage devices to detect any corruption of data.

These functions are highly effective; there is a very low probability that corrupted data can pass a 32-bit CRC check.

CRC functions typically use wide XOR gates to compare the data. The way synthesis tools flatten and factor these XOR gates to implement the logic in FPGA LUTs can greatly impact the area and performance results for the design. XOR gates have a cancellation property that creates an exceptionally large number of reasonable factoring combinations, so synthesis tools cannot always choose the best result by default.

The 6-input ALUT has a significant advantage over 4-input LUTs for these designs. When properly synthesized, CRC processing designs can run at high speeds in devices with 6-input ALUTs.

The following guidelines help you improve the quality of results for CRC designs in Altera devices.

If Performance is Important, Optimize for Speed

Synthesis tools flatten XOR gates to minimize area and depth of levels of logic.

Synthesis tools such as Quartus II integrated synthesis target area optimization by default for these logic structures. Therefore, for more focus on depth reduction, set the synthesis optimization technique to speed.

Flattening for depth sometimes causes a significant increase in area.

Use Separate CRC Blocks Instead of Cascaded Stages

Some designers optimize their CRC designs to use cascaded stages (for example, four stages of 8 bits). In such designs, intermediate calculations are used as required (such as the calculations after 8, 24, or 32 bits) depending on the data width.

This design is not optimal in FPGA devices. The XOR cancellations that can be performed in CRC designs mean that the function does not require all the intermediate calculations to determine the final result. Therefore, forcing the use of intermediate calculations increases the area required to implement the function, as well as increasing the logic depth because of the cascading. It is typically better to create full separate CRC blocks for each data width that you require in the design, and then multiplex them together to choose the appropriate mode at a given time

Use Separate CRC Blocks Instead of Allowing Blocks to Merge

Synthesis tools often attempt to optimize CRC designs by sharing resources and extracting duplicates in two different CRC blocks because of the factoring options in the XOR logic.

The CRC logic allows significant reductions, but this works best when each CRC function is optimized separately. Check for duplicate extraction behavior if you have different CRC functions that are driven by common data signals or that feed the same destination signals.

If you are having problems with the quality of results and you see that two CRC functions are sharing logic, ensure that the blocks are synthesized independently using one of the following methods:

- Define each CRC block as a separate design partition in an incremental compilation design flow.
- Synthesize each CRC block as a separate project in your third-party synthesis tool and then write a separate Verilog Quartus Mapping (**.vqm**) or EDIF netlist file for each.

Related Information

- Quartus II Incremental Compilation on page 3-1

Take Advantage of Latency if Available

If your design can use more than one cycle to implement the CRC functionality, adding registers and retiming the design can help reduce area, improve performance, and reduce power utilization.

If your synthesis tool offers a retiming feature (such as the Quartus II software **Perform gate-level register retiming** option), you can insert an extra bank of registers at the input and allow the retiming feature to move the registers for better results. You can also build the CRC unit half as wide and alternate between halves of the data in each clock cycle.

Save Power by Disabling CRC Blocks When Not in Use

CRC designs are heavy consumers of dynamic power because the logic toggles whenever there is a change in the design.

To save power, use clock enables to disable the CRC function for every clock cycle that the logic is not required. Some designs don’t check the CRC results for a few clock cycles while other logic is performed. It is valuable to disable the CRC function even for this short amount of time.

Use the Device Synchronous Load (sload) Signal to Initialize

The data in many CRC designs must be initialized to 1’s before operation. If your target device supports the use of the `sload` signal, you should use it to set all the registers in your design to 1’s before operation.

To enable use of the `sload` signal, follow the coding guidelines presented in this chapter. You can check the register equations in the Chip Planner to ensure that the signal was used as expected.

If you must force a register implementation using an `sload` signal, you can use low-level device primitives as described in *Designing with Low-Level Primitives User Guide*.

Related Information

- **Secondary Register Control Signals Such as Clear and Clock Enable** on page 12-35
- **Designing with Low-Level Primitives User Guide**

Comparator HDL Guidelines

Synthesis software, including Quartus II integrated synthesis, uses device and context-specific implementation rules for comparators (<, >, or ==) and selects the best one for your design.

This section provides some information about the different types of implementations available and provides suggestions on how you can code your design to encourage a specific implementation.

The == comparator is implemented in general logic cells. The < comparison can be implemented using the carry chain or general logic cells. In devices with 6-input ALUTs, the carry chain is capable of comparing up to three bits per cell. In devices with 4-input LUTs, the capacity is one bit of comparison per cell, which is similar to an add/subtract chain. The carry chain implementation tends to be faster than the general logic on standalone benchmark test cases, but can result in lower performance when it is part of a larger design due to the increased restriction on the Fitter. The area requirement is similar for most input patterns. The synthesis software selects an appropriate implementation based on the input pattern.

If you are using Quartus II integrated synthesis, you can guide the synthesis by using specific coding styles. To select a carry chain implementation explicitly, rephrase your comparison in terms of addition. As a simple example, the following coding style allows the synthesis tool to select the implementation, which is most likely using general logic cells in modern device families:

```
wire [6:0] a,b;
wire alb = a<b;
```

In the following coding style, the synthesis tool uses a carry chain (except for a few cases, such as when the chain is very short or the signals `a` and `b` minimize to the same signal):

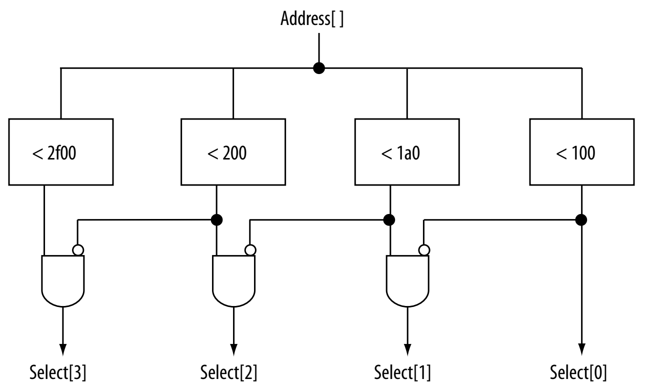
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

This second coding style uses the top bit of the `tmp` signal, which is 1 in twos complement logic if *a* is less than *b*, because the subtraction *a* – *b* results in a negative number.

If you have any information about the range of the input, you have “don’t care” values that you can use to optimize the design. Because this information is not available to the synthesis tool, you can often reduce the device area required to implement the comparator with specific hand implementation of the logic.

You can also check whether a bus value is within a constant range with a small amount of logic area by using the following logic structure . This type of logic occurs frequently in address decoders.

Figure 12-5: Example Logic Structure for Using Comparators to Check a Bus Value Range



Counter HDL Guidelines

Implementing counters in HDL code is easy; they are implemented with an adder followed by registers.

Remember that the register control signals, such as enable (`ena`), synchronous clear (`sclr`), and synchronous load (`sload`), are available. For the best area utilization, ensure that the up/down control or controls are expressed in terms of one addition instead of two separate addition operators.

If you use the following coding style, your synthesis tool may implement two separate carry chains for addition (if it doesn't detect the issue and optimize the logic):

```
out <= count_up ? out + 1 : out - 1;
```

The following coding style requires only one adder along with some other logic:

```
out <= out + (count_up ? 1 : -1);
```

In this case, the coding style better matches the device hardware because there is only one carry chain adder, and the –1 constant logic is implemented in the LUT in front of the adder without adding extra area utilization.

Designing with Low-Level Primitives

Low-level HDL design is the practice of using low-level primitives and assignments to dictate a particular hardware implementation for a piece of logic. Low-level primitives are small architectural building blocks that assist you in creating your design.

With the Quartus II software, you can use low-level HDL design techniques to force a specific hardware implementation that can help you achieve better resource utilization or faster timing results.

Note: Using low-level primitives is an advanced technique to help with specific design challenges, and is optional in the Altera design flow. For many designs, synthesizing generic HDL source code and Altera IP cores give you the best results.

Low-level primitives allow you to use the following types of coding techniques:

- Instantiate the logic cell or `LCELL` primitive to prevent Quartus II integrated synthesis from performing optimizations across a logic cell
- Create carry and cascade chains using `CARRY`, `CARRY_SUM`, and `CASCADE` primitives
- Instantiate registers with specific control signals using `DFF` primitives
- Specify the creation of LUT functions by identifying the LUT boundaries
- Use I/O buffers to specify I/O standards, current strengths, and other I/O assignments
- Use I/O buffers to specify differential pin names in your HDL code, instead of using the automatically-generated negative pin name for each pair

For details about and examples of using these types of assignments, refer to the *Designing with Low-Level Primitives User Guide*.

Related Information

Designing with Low-Level Primitives User Guide

Document Revision History

The following revisions history applies to this chapter.

Table 12-2: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	Added information and reference about ramstyle attribute for sift register inference.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.
2014.08.18	14.0.a10.0	• Added recommendation to use register pipelining to obtain high performance in DSP designs.
2014.06.30	14.0.0	Removed obsolete MegaWizard Plug-In Manager support.
November 2013	13.1.0	Removed HardCopy device support.
June 2012	12.0.0	• Revised section on inserting Altera templates. • Code update for Example 11-51. • Minor corrections and updates.
November 2011	11.1.0	• Updated document template. • Minor updates and corrections.
December 2010	10.1.0	• Changed to new document template. • Updated Unintentional Latch Generation content. • Code update for Example 11-18.

Date	Version	Changes
July 2010	10.0.0	• Added support for mixed-width RAM • Updated support for no_rw_check for inferring RAM blocks • Added support for byte-enable
November 2009	9.1.0	• Updated support for Controlling Inference and Implementation in Device RAM Blocks • Updated support for Shift Registers
March 2009	9.0.0	• Corrected and updated several examples • Added support for Arria II GX devices • Other minor changes to chapter
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	Updates for the Quartus II software version 8.0 release, including: • Added information to "RAM • Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code" on page 6–13 • Added information to "Avoid Unsupported Reset and Control Conditions" on page 6–14 • Added information to "Check Read-During-Write Behavior" on page 6–16 • Added two new examples to "ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code" on page 6–28: Example 6–24 and Example 6–25 • Added new section: "Clock Multiplexing" on page 6–46 • Added hyperlinks to references within the chapter • Minor editorial updates

Related Information

Quartus II Handbook Archive

Managing Metastability with the Quartus II Software 13

2014.12.15

QII5V1 Subscribe Send Feedback

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

All registers in digital devices, such as FPGAs, have defined signal-timing requirements that allow each register to correctly capture data at its input ports and produce an output signal. To ensure reliable operation, the input to a register must be stable for a minimum amount of time before the clock edge (register setup time or t_{SU}) and a minimum amount of time after the clock edge (register hold time or t_H). The register output is available after a specified clock-to-output delay (t_{CO}).

If the data violates the setup or hold time requirements, the output of the register might go into a metastable state. In a metastable state, the voltage at the register output hovers at a value between the high and low states, which means the output transition to a defined high or low state is delayed beyond the specified t_{CO}. Different destination registers might capture different values for the metastable signal, which can cause the system to fail.

In synchronous systems, the input signals must always meet the register timing requirements, so that metastability does not occur. Metastability problems commonly occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the signal can arrive at any time relative to the destination clock.

The MTBF due to metastability is an estimate of the average time between instances when metastability could cause a design failure. A high MTBF (such as hundreds or thousands of years between metastability failures) indicates a more robust design. You should determine an acceptable target MTBF in the context of your entire system and taking in account that MTBF calculations are statistical estimates.

The metastability MTBF for a specific signal transfer, or all the transfers in a design, can be calculated using information about the design and the device characteristics. Improving the metastability MTBF for your design reduces the chance that signal transfers could cause metastability problems in your device.

The Quartus II software provides analysis, optimization, and reporting features to help manage metastability in Altera designs. These metastability features are supported only for designs constrained with the Quartus II Timing Analyzer. Both typical and worst-case MBTF values are generated for select device families.

Related Information

- **Understanding Metastability in FPGAs**
  For more information about metastability due to signal synchronization, its effects in FPGAs, and how MTBF is calculated

- **Reliability Report**
  For information about Altera device reliability

Metastability Analysis in the Quartus II Software

When a signal transfers between circuitry in unrelated or asynchronous clock domains, the first register in the new clock domain acts as a synchronization register.

To minimize the failures due to metastability in asynchronous signal transfers, circuit designers typically use a sequence of registers (a synchronization register chain or synchronizer) in the destination clock domain to resynchronize the signal to the new clock domain and allow additional time for a potentially metastable signal to resolve to a known value. Designers commonly use two registers to synchronize a new signal, but a standard of three registers provides better metastability protection.

The timing analyzer can analyze and report the MTBF for each identified synchronizer that meets its timing requirements, and can generate an estimate of the overall design MTBF. The software uses this information to optimize the design MTBF, and you can use this information to determine whether your design requires longer synchronizer chains.

Related Information

- **Metastability and MTBF Reporting** on page 13-4
- **MTBF Optimization** on page 13-7

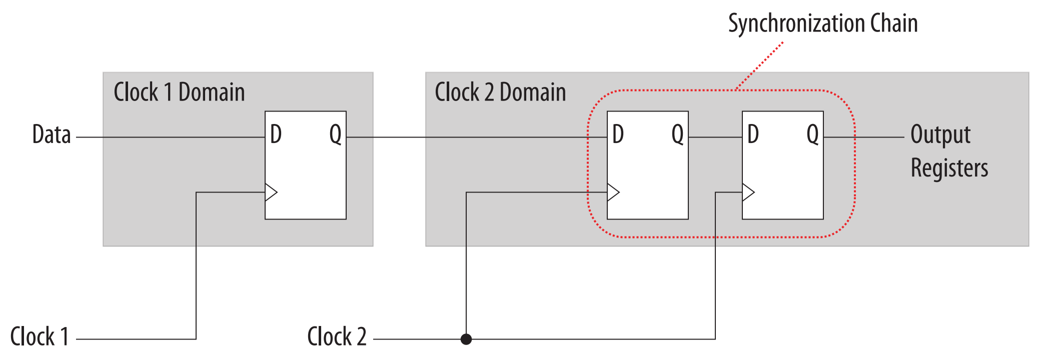
Synchronization Register Chains

A synchronization register chain, or synchronizer, is defined as a sequence of registers that meets the following requirements:

- The registers in the chain are all clocked by the same clock or phase-related clocks.
- The first register in the chain is driven asynchronously or from an unrelated clock domain.
- Each register fans out to only one register, except the last register in the chain.

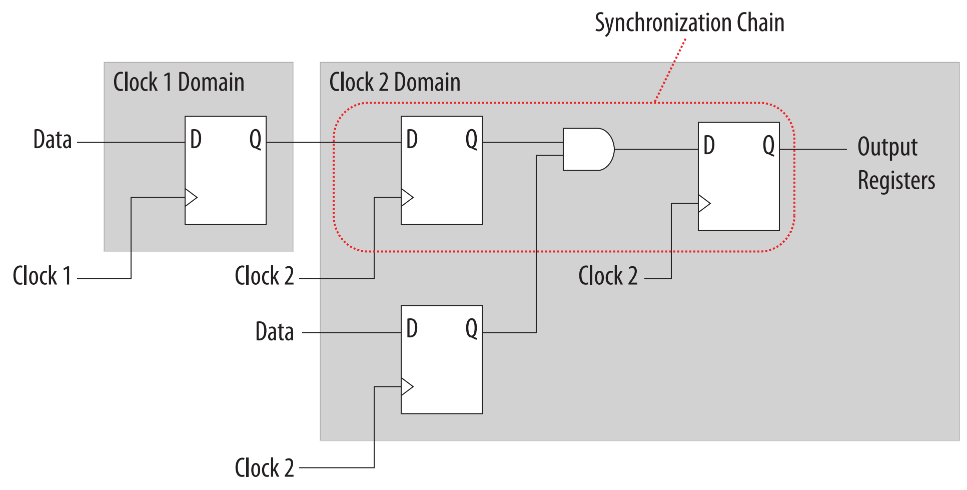
The length of the synchronization register chain is the number of registers in the synchronizing clock domain that meet the above requirements. The figure shows a sample two-register synchronization chain.

Figure 13-1: Sample Synchronization Register Chain



The path between synchronization registers can contain combinational logic as long as all registers of the synchronization register chain are in the same clock domain. The figure shows an example of a synchronization register chain that includes logic between the registers.

Figure 13-2: Sample Synchronization Register Chain Containing Logic



The timing slack available in the register-to-register paths of the synchronizer allows a metastable signal to settle, and is referred to as the available settling time. The available settling time in the MTBF calculation for a synchronizer is the sum of the output timing slacks for each register in the chain. Adding available settling time with additional synchronization registers improves the metastability MTBF.

Related Information

How Timing Constraints Affect Synchronizer Identification and Metastability Analysis on page 13-3

Identifying Synchronizers for Metastability Analysis

The first step in enabling metastability MTBF analysis and optimization in the Quartus II software is to identify which registers are part of a synchronization register chain. You can apply synchronizer identification settings globally to automatically list possible synchronizers with the **Synchronizer identification** option on the **Timing Analyzer** page in the **Settings** dialog box.

Synchronization chains are already identified within most Altera intellectual property (IP) cores.

Related Information

Identifying Synchronizers for Metastability
For more information about how to enable metastability MTBF analysis and optimization in the Quartus II software, and more detailed descriptions of the synchronizer identification settings

How Timing Constraints Affect Synchronizer Identification and Metastability Analysis

The timing analyzer can analyze metastability MTBF only if a synchronization chain meets its timing requirements. The metastability failure rate depends on the timing slack available in the synchronizer's register-to-register connections, because that slack is the available settling time for a potential metastable signal. Therefore, you must ensure that your design is correctly constrained with the real application frequency requirements to get an accurate MTBF report.

In addition, the **Auto** and **Forced If Asynchronous** synchronizer identification options use timing constraints to automatically detect the synchronizer chains in the design. These options check for signal transfers between circuitry in unrelated or asynchronous clock domains, so clock domains must be related correctly with timing constraints.

The timing analyzer views input ports as asynchronous signals unless they are associated correctly with a clock domain. If an input port fans out to registers that are not acting as synchronization registers, apply a `set_input_delay` constraint to the input port; otherwise, the input register might be reported as a synchronization register. Constraining a synchronous input port with a `set_max_delay` constraint for a setup (t_{SU}) requirement does not prevent synchronizer identification because the constraint does not associate the input port with a clock domain.

Instead, use the following command to specify an input setup requirement associated with a clock:

`set_input_delay -max -clock` *<clock name> <latch – launch – tsu requirement> <input port name>*

Registers that are at the end of false paths are also considered synchronization registers because false paths are not timing-analyzed. Because there are no timing requirements for these paths, the signal may change at any point, which may violate the t_{SU} and t_H of the register. Therefore, these registers are identified as synchronization registers. If these registers are not used for synchronization, you can turn off synchronizer identification and analysis. To do so, set **Synchronizer Identification** to **Off** for the first synchronization register in these register chains.

Metastability and MTBF Reporting

The Quartus II software reports the metastability analysis results in the Compilation Report and Timing Analyzer reports.

The MTBF calculation uses timing and structural information about the design, silicon characteristics, and operating conditions, along with the data toggle rate.

If you change the **Synchronizer Identification** settings, you can generate new metastability reports by rerunning the timing analyzer. However, you should rerun the Fitter first so that the registers identified with the new setting can be optimized for metastability MTBF.

Related Information

- **Metastability Reports** on page 13-4
- **MTBF Optimization** on page 13-7
- **Synchronizer Data Toggle Rate in MTBF Calculation** on page 13-6
- **Understanding Metastability in FPGAs**
  For more information about how metastability MTBF is calculated

Metastability Reports

Metastability reports provide summaries of the metastability analysis results. In addition to the MTBF Summary and Synchronizer Summary reports, the Timing Analyzer tool reports additional statistics in a report for each synchronizer chain.

Note: If the design uses only the **Auto Synchronizer Identification** setting, the reports list likely synchronizers but do not report MTBF. To obtain an MTBF for each register chain, force identification of synchronization registers.

Note: If the synchronizer chain does not meet its timing requirements, the reports list identified synchronizers but do not report MTBF. To obtain MTBF calculations, ensure that the design is properly constrained and that the synchronizer meets its timing requirements.

Related Information

- **Identifying Synchronizers for Metastability Analysis** on page 13-3
- **How Timing Constraints Affect Synchronizer Identification and Metastability Analysis** on page 13-3
- **Viewing Metastability Reports**
  For more information about how to access metastability reports in the Quartus II software

MTBF Summary Report

The MTBF Summary reports an estimate of the overall robustness of cross-clock domain and asynchronous transfers in the design. This estimate uses the MTBF results of all synchronization chains in the design to calculate an MTBF for the entire design.

Typical and Worst-Case MTBF of Design

The MTBF Summary Report shows the **Typical MTBF of Design** and the **Worst-Case MTBF of Design** for supported fully-characterized devices. The typical MTBF result assumes typical conditions, defined as nominal silicon characteristics for the selected device speed grade, as well as nominal operating conditions. The worst case MTBF result uses the worst case silicon characteristics for the selected device speed grade.

When you analyze multiple timing corners in the timing analyzer, the MTBF calculation may vary because of changes in the operating conditions, and the timing slack or available metastability settling time. Altera recommends running multi-corner timing analysis to ensure that you analyze the worst MTBF results, because the worst timing corner for MTBF does not necessarily match the worst corner for timing performance.

Related Information

Timing Analyzer page

Synchronizer Chains

The MTBF Summary report also lists the **Number of Synchronizer Chains Found** and the length of the **Shortest Synchronizer Chain**, which can help you identify whether the report is based on accurate information.

If the number of synchronizer chains found is different from what you expect, or if the length of the shortest synchronizer chain is less than you expect, you might have to add or change **Synchronizer Identification** settings for the design. The report also provides the **Worst Case Available Settling Time**, defined as the available settling time for the synchronizer with the worst MTBF.

You can use the reported **Fraction of Chains for which MTBFs Could Not be Calculated** to determine whether a high proportion of chains are missing in the metastability analysis. A fraction of 1, for example, means that MTBF could not be calculated for any chains in the design. MTBF is not calculated if you have not identified the chain with the appropriate **Synchronizer identification** option, or if paths are not timing-analyzed and therefore have no valid slack for metastability analysis. You might have to correct your timing constraints to enable complete analysis of the applicable register chains.

Increasing Available Settling Time

The MTBF Summary report specifies how an increase of 100ps in available settling time increases the MTBF values. If your MTBF is not satisfactory, this metric can help you determine how much extra slack would be required in your synchronizer chain to allow you to reach the desired design MTBF.

Synchronizer Summary Report

The **Synchronizer Summary** lists the synchronization register chains detected in the design depending on the Synchronizer Identification setting.

The **Source Node** is the register or input port that is the source of the asynchronous transfer. The **Synchronization Node** is the first register of the synchronization chain. The **Source Clock** is the clock domain of the source node, and the **Synchronization Clock** is the clock domain of the synchronizer chain.

This summary reports the calculated **Worst-Case MTBF**, if available, and the **Typical MTBF**, for each appropriately identified synchronization register chain that meets its timing requirement.

Related Information

Synchronizer Chain Statistics Report in the Timing Analyzer on page 13-6

Synchronizer Chain Statistics Report in the Timing Analyzer

The timing analyzer provides an additional report for each synchronizer chain.

The **Chain Summary** tab matches the Synchronizer Summary information described in Synchronizer Summary Report, while the **Statistics** tab adds more details, including whether the **Method of Synchronizer Identification** was **User Specified** (with the **Forced if Asynchronous** or **Forced** settings for the **Synchronizer Identification** setting), or **Automatic** (with the **Auto** setting). The **Number of Synchronization Registers in Chain** report provides information about the parameters that affect the MTBF calculation, including the **Available Settling Time** for the chain and the **Data Toggle Rate Used in MTBF Calculation**.

The following information is also included to help you locate the chain in your design:

- **Source Clock** and **Asynchronous Source** node of the signal.
- **Synchronization Clock** in the destination clock domain.
- Node names of the **Synchronization Registers** in the chain.

Related Information

Synchronizer Data Toggle Rate in MTBF Calculation on page 13-6

Synchronizer Data Toggle Rate in MTBF Calculation

The MTBF calculations assume the data being synchronized is switching at a toggle rate of 12.5% of the source clock frequency. That is, the arriving data is assumed to switch once every eight source clock cycles.

If multiple clocks apply, the highest frequency is used. If no source clocks can be determined, the data rate is taken as 12.5% of the synchronization clock frequency.

If you know an approximate rate at which the data changes, specify it with the **Synchronizer Toggle Rate** assignment in the Assignment Editor. You can also apply this assignment to an entity or the entire design. Set the data toggle rate, in number of transitions per second, on the first register of a synchronization chain. The timing analyzer takes the specified rate into account when computing the MTBF of that particular register chain. If a data signal never toggles and does not affect the reliability of the design, you can set the **Synchronizer Toggle Rate** to **0** for the synchronization chain so the MTBF is not reported. To apply the assignment with Tcl, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_TOGGLE_RATE <toggle rate in transitions/
second> -to <register name>
```

In addtion to **Synchronizer Toggle Rate,** there are two other assignments associated with toggle rates, which are not used for metastability MTBF calculations. The I/O Maximum Toggle Rate is only used for pins, and specifies the worst-case toggle rates used for signal integrity purposes. The Power Toggle Rate assignment is used to specify the expected time-averaged toggle rate, and is used by the PowerPlay Power Analyzer to estimate time-averaged power consumption.

MTBF Optimization

In addition to reporting synchronization register chains and MTBF values found in the design, the Quartus II software can also protect these registers from optimizations that might negatively impact MTBF and can optimize the register placement and routing if the MTBF is too low.

Synchronization register chains must first be explicitly identified as synchronizers. Altera recommends that you set **Synchronizer Identification** to **Forced If Asynchronous** for all registers that are part of a synchronizer chain.

Optimization algorithms, such as register duplication and logic retiming in physical synthesis, are not performed on identified synchronization registers. The Fitter protects the number of synchronization registers specified by the **Synchronizer Register Chain Length** option.

In addition, the Fitter optimizes identified synchronizers for improved MTBF by placing and routing the registers to increase their output setup slack values. Adding slack in the synchronizer chain increases the available settling time for a potentially metastable signal, which improves the chance that the signal resolves to a known value, and exponentially increases the design MTBF. The Fitter optimizes the number of synchronization registers specified by the **Synchronizer Register Chain Length** option.

Metastability optimization is **on** by default. To view or change the **Optimize Design for Metastability** option, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**. To turn the optimization on or off with Tcl, use the following command:

```
set_global_assignment -name OPTIMIZE_FOR_METASTABILITY <ON|OFF>
```

Related Information

Identifying Synchronizers for Metastability Analysis on page 13-3

Synchronization Register Chain Length

The **Synchronization Register Chain Length** option specifies how many registers should be protected from optimizations that might reduce MTBF for each register chain, and controls how many registers should be optimized to increase MTBF with the **Optimize Design for Metastability** option.

For example, if the **Synchronization Register Chain Length** option is set to **2**, optimizations such as register duplication or logic retiming are prevented from being performed on the first two registers in all identified synchronization chains. The first two registers are also optimized to improve MTBF when the **Optimize Design for Metastability** option is turned on.

The default setting for the **Synchronization Register Chain Length** option is **2**. The first register of a synchronization chain is always protected from operations that might reduce MTBF, but you should set the protection length to protect more of the synchronizer chain. Altera recommends that you set this option to the maximum length of synchronization chains you have in your design so that all synchronization registers are preserved and optimized for MTBF.

Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** to change the global **Synchronization Register Chain Length** option.

You can also set the **Synchronization Register Chain Length** on a node or an entity in the Assignment Editor. You can set this value on the first register in a synchronization chain to specify how many registers to protect and optimize in this chain. This individual setting is useful if you want to protect and optimize extra registers that you have created in a specific synchronization chain that has low MTBF, or optimize less registers for MTBF in a specific chain where the maximum frequency or timing performance is not being met. To make the global setting with Tcl, use the following command:

```
set_global_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH <number of
registers>
```

To apply the assignment to a design instance or the first register in a specific chain with Tcl, use the following command:

```
set_instance_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH <number of
registers> -to <register or instance name>
```

Reducing Metastability Effects

You can check your design's metastability MTBF in the Metastability Summary report, and determine an acceptable target MTBF given the context of your entire system and the fact that MTBF calculations are statistical estimates. A high metastability MTBF (such as hundreds or thousands of years between metastability failures) indicates a more robust design.

This section provides guidelines to ensure complete and accurate metastability analysis, and some suggestions to follow if the Quartus II metastability reports calculate an unacceptable MTBF value. The Timing Optimization Advisor (available from the Tools menu) gives similar suggestions in the Metastability Optimization section.

Related Information

Metastability Reports on page 13-4

Apply Complete System-Centric Timing Constraints for the Timing Analyzer

To enable the Quartus II metastability features, make sure that the timing analyzer is used for timing analysis.

Ensure that the design is fully timing constrained and that it meets its timing requirements. If the synchronization chain does not meet its timing requirements, MTBF cannot be calculated. If the clock domain constraints are set up incorrectly, the signal transfers between circuitry in unrelated or asynchronous clock domains might be identified incorrectly.

Use industry-standard system-centric I/O timing constraints instead of using FPGA-centric timing constraints.

You should use `set_input_delay` constraints in place of `set_max_delay` constraints to associate each input port with a clock domain to help eliminate false positives during synchronization register identification.

Related Information

How Timing Constraints Affect Synchronizer Identification and Metastability Analysis on page 13-3

Force the Identification of Synchronization Registers

Use the guidelines in Identifying Synchronizers for Metastability Analysis to ensure the software reports and optimizes the appropriate register chains.

You should identify synchronization registers with the **Synchronizer Identification** set to **Forced If Asynchronous** in the Assignment Editor. If there are any registers that the software detects as synchronous but you want to be analyzed for metastability, apply the **Forced** setting to the first synchronizing register. Set **Synchronizer Identification** to **Off** for registers that are not synchronizers for asynchronous signals or unrelated clock domains.

To help you find the synchronizers in your design, you can set the global **Synchronizer Identification** setting on the **Timing Analyzer** page of the **Settings** dialog box to **Auto** to generate a list of all the possible synchronization chains in your design.

Related Information

Identifying Synchronizers for Metastability Analysis on page 13-3

Set the Synchronizer Data Toggle Rate

The MTBF calculations assume the data being synchronized is switching at a toggle rate of 12.5% of the source clock frequency.

To obtain a more accurate MTBF for a specific chain or all chains in your design, set the **Synchronizer Toggle Rate**.

Related Information

Synchronizer Data Toggle Rate in MTBF Calculation on page 13-6

Optimize Metastability During Fitting

Ensure that the **Optimize Design for Metastability** setting is turned on.

Related Information

MTBF Optimization on page 13-7

Increase the Length of Synchronizers to Protect and Optimize

Increase the Synchronizer Chain Length parameter to the maximum length of synchronization chains in your design. If you have synchronization chains longer than 2 identified in your design, you can protect the entire synchronization chain from operations that might reduce MTBF and allow metastability optimizations to improve the MTBF.

Related Information

Synchronization Register Chain Length on page 13-7

Set Fitter Effort to Standard Fit instead of Auto Fit

If your design MTBF is too low after following the other guidelines, you can try increasing the Fitter effort to perform more metastability optimization. The default **Auto Fit** setting reduces the Fitter's effort after meeting the design's timing and routing requirements to reduce compilation time.

This effort reduction can result in less metastability optimization if the timing requirements are easy to meet. If **Auto Fit** reduces the Fitter's effort during your design compilation, setting the Fitter effort to **Standard Fit** might improve the design's MTBF results. To modify the **Fitter Effort**, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**.

Increase the Number of Stages Used in Synchronizers

Designers commonly use two registers in a synchronization chain to minimize the occurrence of metastable events, and a standard of three registers provides better metastability protection. However, synchronization chains with two or even three registers may not be enough to produce a high enough MTBF when the design runs at high clock and data frequencies.

If a synchronization chain is reported to have a low MTBF, consider adding an additional register stage to your synchronization chain. This additional stage increases the settling time of the synchronization chain, allowing more opportunity for the signal to resolve to a known state during a metastable event. Additional settling time increases the MTBF of the chain and improves the robustness of your design. However, adding a synchronization stage introduces an additional stage of latency on the signal.

If you use the Altera FIFO IP core with separate read and write clocks to cross clock domains, increase the metastability protection (and latency) for better MTBF. In the DCFIFO parameter editor, choose the **Best metastability protection, best fmax, unsynchronized clocks** option to add three or more synchronization stages. You can increase the number of stages to more than three using the **How many sync stages?** setting.

Select a Faster Speed Grade Device

The design MTBF depends on process parameters of the device used. Faster devices are less susceptible to metastability issues. If the design MTBF falls significantly below the target MTBF, switching to a faster speed grade can improve the MTBF substantially.

Scripting Support

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

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp r
```

Related Information

- Tcl Scripting
  For more information about Tcl scripting
- Quartus II Settings File Reference Manual
  For more information about settings and constraints in the Quartus II software

- **Command-Line Scripting**
  For more information about command-line scripting
- **About Quartus II Scripting**
  For more information about command-line scripting

Identifying Synchronizers for Metastability Analysis

To apply the global Synchronizer Identification assignment, use the following command:

```
set_global_assignment -name SYNCHRONIZER_IDENTIFICATION <OFF|AUTO|"FORCED IF
ASYNCHRONOUS">
```

To apply the **Synchronizer Identification** assignment to a specific register or instance, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_IDENTIFICATION <AUTO|"FORCED IF ASYNCHRO-
NOUS"|FORCED|OFF> -to <register or instance name>
```

Synchronizer Data Toggle Rate in MTBF Calculation

To specify a toggle rate for MTBF calculations as described on page **Synchronizer Data Toggle Rate in MTBF Calculation**, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_TOGGLE_RATE <toggle rate in transitions/
second> -to <register name>
```

Related Information

Synchronizer Data Toggle Rate in MTBF Calculation on page 13-6

report_metastability and Tcl Command

If you use a command-line or scripting flow, you can generate the metastability analysis reports described in **Metastability Reports** outside of the Quartus II and user interfaces.

The table describes the options for the `report_metastability` and Tcl command.

Table 13-1: report_metastabilty Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type — either ***.txt** or ***.html**.
`-panel_name` <name>	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report be sent to the standard output, via messages. This option is required only if you have selected another output format, such as a file, and would also like to receive messages.

Related Information

Metastability Reports on page 13-4

MTBF Optimization

To ensure that metastability optimization described on page MTBF Optimization is turned on (or to turn it off), use the following command:

```
set_global_assignment -name OPTIMIZE_FOR_METASTABILITY <ON|OFF>
```

Related Information

MTBF Optimization on page 13-7

Synchronization Register Chain Length

To globally set the number of registers in a synchronization chain to be protected and optimized as described on page Synchronization Register Chain Length, use the following command:

```
set_global_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH <number of
registers>
```

To apply the assignment to a design instance or the first register in a specific chain, use the following command:

```
set_instance_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH <number of
registers> -to <register or instance name>
```

Related Information

Synchronization Register Chain Length on page 13-7

Managing Metastability

Altera's Quartus II software provides industry-leading analysis and optimization features to help you manage metastability in your FPGA designs. Set up your Quartus II project with the appropriate constraints and settings to enable the software to analyze, report, and optimize the design MTBF. Take advantage of these features in the Quartus II software to make your design more robust with respect to metastability.

Document Revision History

Table 13-2: Document Revision History

Date	Version	Changes
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.
June 2014	14.0.0	Updated formatting.

Date	Version	Changes
June 2012	12.0.0	Removed survey link.
November 2011	10.0.2	Template update.
December 2010	10.0.1	Changed to new document template.
July 2010	10.0.0	Technical edit.
November 2009	9.1.0	Clarified description of synchronizer identification settings. Minor changes to text and figures throughout document.
March 2009	9.0.0	Initial release.

Related Information

Quartus II Handbook Archive
For previous versions of the Quartus II Handbook

Best Practices for Incremental Compilation Partitions and Floorplan Assignments 14

2015.05.04

QII5V1 Subscribe Send Feedback

About Incremental Compilation and Floorplan Assignments

This manual provides guidelines to help you partition your design to take advantage of Quartus® II incremental compilation, and to help you create a design floorplan using LogicLock™ regions when they are recommended to support the compilation flow.

The Quartus II incremental compilation feature allows you to partition a design, compile partitions separately, and reuse results for unchanged partitions. Incremental compilation provides the following benefits:

- Reduces compilation times by an average of 75% for large design changes
- Preserves performance for unchanged design blocks
- Provides repeatable results and reduces the number of compilations
- Enables team-based design flows

Related Information

Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation on page 3-1

Incremental Compilation online help

Incremental Compilation Overview

Quartus II incremental compilation is an optional compilation flow that enhances the default Quartus II compilation. If you do not partition your design for incremental compilation, your design is compiled using the default “flat” compilation flow.

To prepare your design for incremental compilation, you first determine which logical hierarchy boundaries should be defined as separate partitions in your design, and ensure your design hierarchy and source code is set up to support this partitioning. You can then create design partition assignments in the Quartus II software to specify which hierarchy blocks are compiled independently as partitions (including empty partitions for missing or incomplete logic blocks).

During compilation, Quartus II Analysis & Synthesis and the Fitter create separate netlists for each partition. Netlists are internal post-synthesis and post-fit database representations of your design.

In subsequent compilations, you can select which netlist to preserve for each partition. You can either reuse the synthesis or fitting netlist, or instruct the Quartus II software to resynthesize the source files. You can also use compilation results exported from another Quartus II project.

When you make changes to your design, the Quartus II software recompiles only the designated partitions and merges the new compilation results with existing netlists for other partitions, according to the degree of results preservation you set with the netlist for each design partition.

In some cases, Altera recommends that you create a design floorplan with placement assignments to constrain parts of the design to specific regions of the device.

You must use the partial reconfiguration (PR) feature in conjunction with incremental compilation for Stratix® V device families. Partial reconfiguration allows you to reconfigure a portion of the FPGA dynamically, while the remainder of the device continues to operate as intended.

Related Information

- **Introduction to Design Floorplans** on page 14-36
- **Using the Incremental Compilation Design Flow online help**
  Step-by-step information about using incremental compilation to recompile only changed parts of your design
- **Design Planning for Partial Reconfiguration documentation** on page 4-1

The Partial Reconfiguration (PR) feature in the Quartus II software allows you to reconfigure a portion of the FPGA dynamically, while the remainder of the device continues to operate. The Quartus II software supports the PR feature for the Altera® Stratix® V device family.

Recommendations for the Netlist Type

For subsequent compilations, you specify which post-compilation netlist you want to use with the netlist type for each partition.

Use the following general guidelines to set the netlist type for each partition:

- **Source File**—Use this setting to resynthesize the source code (with any new assignments, and replace any previous synthesis or Fitter results).
  - If you modify the design source, the software automatically resynthesizes the partitions with the appropriate netlist type, which makes the **Source File** setting optional in this case.
  - Most assignments do not trigger an automatic recompilation, so you must set the netlist type to **Source File** to compile the source files with new assignments or constraints that affect synthesis.
- **Post-Synthesis** (default)—Use this setting to re-fit the design (with any new Fitter assignments), but preserve the synthesis results when the source files have not changed. If it is difficult to meet the required timing performance, you can use this setting to allow the Fitter the most flexibility in placement and routing. This setting does not reduce compilation time as much as the **Post-Fit** setting or preserve timing performance from the previous compilation.
- **Post-Fit**—Use this setting to preserve Fitter and performance results when the source files have not changed. This setting reduces compilation time the most, and preserves timing performance from the previous compilation.
- **Post-Fit with Fitter Preservation Level set to Placement**—Use the **Advanced Fitter Preservation Level** setting on the **Advanced** tab in the **Design Partition Properties** dialog box to allow more flexibility and find the best routing for all partitions given their placement.

The Quartus II software Rapid Recompile feature instructs the Compiler to reuse the compatible compilation results if most of the design has not changed since the last compilation. This feature reduces compilation time and preserves performance when there are small and isolated design changes within a partition, and works with all netlist type settings. With this feature, you do not have control over which parts of the design are recompiled; the Compiler determines which parts of the design must be recompiled.

Design Flows Using Incremental Compilation

The Quartus II incremental compilation feature supports various design flows. Your design flow affects design optimization and the amount of design planning required to obtain optimal results.

Using Standard Flow

In the standard incremental compilation flow, the top-level design is divided into partitions, which can be compiled and optimized together in one Quartus II project. If another team member or IP provider is developing source code for the top-level design, they can functionally verify their partition independently, and then simply provide the partition's source code to the project lead for integration into the top-level design. If the project lead wants to compile the top-level design when source code is not yet complete for a partition, they can create an empty placeholder for the partition until the code is ready to be added to the top-level design.

Compiling all design partitions in a single Quartus II project ensures that all design logic is compiled with a consistent set of assignments, and allows the software to perform global placement and routing optimizations. Compiling all design logic together is beneficial for FPGA design flows because all parts of the design must use the same shared set of device resources. Therefore, it is often easier to ensure good quality of results when partitions are developed within a single top-level Quartus II project.

Using Team-Based Flow

In the team-based incremental compilation flow, you can design and optimize partitions by accessing the top-level project from a shared source control system or creating copies of the top-level Quartus II project framework. As development continues, designers export their partition so that the post-synthesis netlist or post-fitting results can be integrated into the top-level design.

Using Third-Party IP Delivery Flow

If required for third-party IP delivery, or in cases where designers cannot access a shared or copied top-level project framework, you can create and compile a design partition logic in isolation and export a partition that is included in the top-level project. If this type of design flow is necessary, planning and rigorous design guidelines might be required to ensure that designers have a consistent view of project assignments and resource allocations. Therefore, developing partitions in completely separate Quartus II projects can be more challenging than having all source code within one project or developing design partitions within the same top-level project framework.

Combining Design Flows

You can also combine design flows and use exported partitions only when it is necessary to support your design environment. For example, if the top-level design includes one or more design blocks that will be optimized by remote designers or IP providers, you can integrate those blocks into the reserved partitions in the top-level design when the code is complete, but also have other partitions that will be developed within the top-level design.

If any partitions are developed independently, the project lead must ensure that top-level constraints (such as timing constraints, any relevant floorplan or pin assignments, and optimization settings) are consistent with those used by all designers.

Project Management in Team-Based Design Flows

If possible, each team member should work within the same top-level project framework. Using the same project framework amongst team members ensures that designers have the settings and constraints needed for their partition and allows designers to analyze how their design block interacts with other partitions in the top-level design.

Using a Source Control System

In a team-based environment where designers have access to the project through source control software, each designer can use project files as read-only and develop their partition within the source control system. As designers check in their completed partitions, other team members can see how their partitions interact.

Using a Copy of the Top-Level Project

If designers do not have access to a source control system, the project lead can provide each designer with a copy of the top-level project framework to use as they develop their partitions. In both cases, each designer exports their completed design as a partition, and then the project lead integrates the partition into the top-level design. The project lead can choose to use only the post-synthesis netlist and rerun placement and routing, or to use the post-fitting results to preserve the placement and routing results from the other designer's projects. Using post-synthesis partitions gives the Fitter the most flexibility and is likely to achieve a good result for all partitions, but if one partition has difficultly meeting timing, the designer can choose to preserve their successful fitting results.

Using a Separate Project

Alternatively, designers can use their own Quartus II project for their independent design block. You might use this design flow if a designer, such as a third-party IP provider, does not have access to the entire top-level project framework. In this case, each designer must create their own project with all the relevant assignments and constraints. This type of design flow requires more planning and rigorous design guidelines. If the project lead plans to incorporate the post-fitting compilation results for the partition, this design flow requires especially careful planning to avoid resource conflicts.

Using Scripts

The project lead also has the option to generate design partition scripts to manage resource and timing budgets in the top-level design when partitions are developed outside the top-level project framework. Scripts make it easier for designers of independent Quartus II projects to follow instructions from the project lead. The Quartus II design partition scripts feature creates Tcl scripts or **.tcl** files and makefiles that an independent designer can run to set up an independent Quartus II project.

Related Information

Generating Design Partition Scripts for Project Management online help

Using Constraints

If designers create Quartus II assignments or timing constraints for their partitions, they must ensure that the constraints are integrated into the top-level design. If partition designers use the same top-level project framework (and design hierarchy), the constraints or Synopsys Design Constraints File (**.sdc**) can be easily copied or included in the top-level design. If partition designers use a separate Quartus II project with a different design hierarchy, they must ensure that constraints are applied to the appropriate level of hierarchy in the top-level design, and design the **.sdc** for easy delivery to the project lead.

Related Information

Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery on page 14-32

Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation on page 3-1
Information about the different types of incremental design flows and example applications, as well as documented restrictions and limitations

Why Plan Partitions and Floorplan Assignments?

Incremental design flows typically require more planning than flat compilations, and require you to be more rigorous about following good design practices. For example, you might need to structure your source code or design hierarchy to ensure that logic is grouped correctly for optimization. It is easier to implement the correct logic grouping early in the design cycle than to restructure the code later.

Planning involves setting up the design logic for partitioning and may also involve planning placement assignments to create a floorplan. Not all design flows require floorplan assignments. If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation easier. Poor partition or floorplan assignments can worsen design area utilization and performance and make timing closure more difficult.

As FPGA devices get larger and more complex, following good design practices become more important for all design flows. Adhering to recommended synchronous design practices makes designs more robust and easier to debug. Using an incremental compilation flow adds additional steps and requirements to your project, but can provide significant benefits in design productivity by preserving the performance of critical blocks and reducing compilation time.

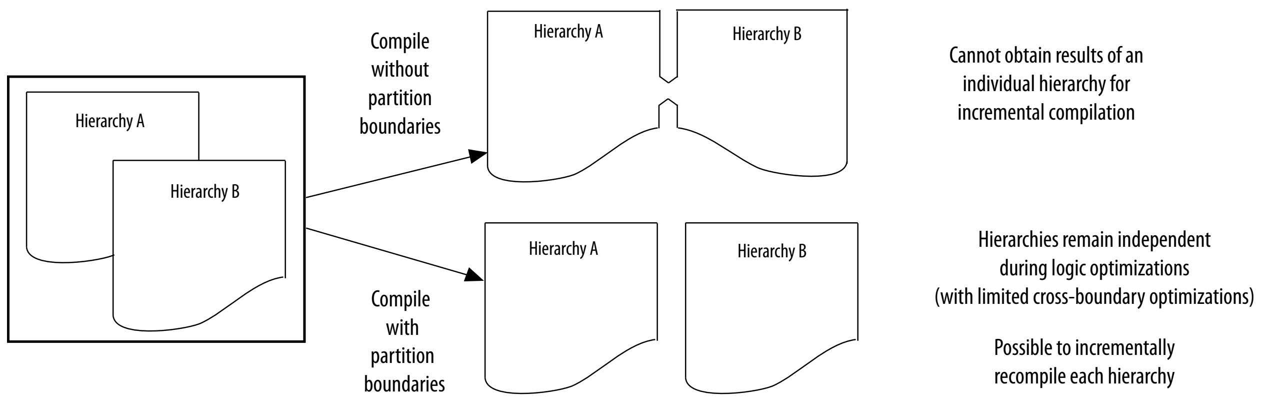
Related Information

Introduction to Design Floorplans on page 14-36

Partition Boundaries and Optimization

The logical hierarchical boundaries between partitions are treated as hard boundaries for logic optimization (except for some limited cross-boundary optimizations) to allow the software to size and place each partition independently. The figure shows the effects of partition boundaries during logic optimization.

Figure 14-1: Effects of Partition Boundaries During Logic Optimization


Hierarchy A
Hierarchy B
Compile without partition boundaries
Hierarchy A
Hierarchy B
Cannot obtain results of an individual hierarchy for incremental compilation
Compile with partition boundaries
Hierarchy A
Hierarchy B
Hierarchies remain independent during logic optimizations (with limited cross-boundary optimizations)
Possible to incrementally recompile each hierarchy

Merging Partitions

You can use the **Merge** command in the Design Partitions window to combine hierarchical partitions into a single partition, as long as they share the same immediate parent partition. Merging partitions allows additional optimizations for partition I/O ports that connect between or feed more than one of the merged hierarchical design blocks.

When partitions are placed together, the Fitter can perform placement optimizations on the design as a whole to optimize the placement of cross-boundary paths. However, the Fitter can never perform logic optimizations such as physical synthesis across the partition boundary. If partitions are fit separately in different projects, or if some partitions use previous post-fitting results, the Fitter does not place and route the entire cross-boundary path at the same time and cannot fully optimize placement across the partition boundaries. Good design partitions can be placed independently because cross-partition paths are not the critical timing paths in the design.

Resource Utilization

There are possible timing performance utilization effects due to partitioning and creating a floorplan. Not all designs encounter these issues, but you should consider these effects if a flat version of your design is very close to meeting its timing requirements, or is close to using all the device resources, before adding partition or floorplan assignments:

- Partitions can increase resource utilization due to cross-boundary optimization limitations if the design does not follow partitioning guidelines. Floorplan assignments can also increase resource utilization because regions can lead to unused logic. If your device is full with the flat version of your design, you can focus on creating partitions and floorplan assignments for timing-critical or often-changing blocks to benefit most from incremental compilation.
- Partitions and floorplan assignments might increase routing utilization compared to a flat design. If long compilation times are due to routing congestion, you might not be able to use the incremental flow to reduce compilation time. Review the Fitter messages to check how much time is spent during routing optimizations to determine the percentage of routing utilization. When routing is difficult, you can use incremental compilation to lock the routing for routing-critical blocks only (with other partitions empty), and then compile the rest of the design after the critical blocks meets their requirements.
- Partitions can reduce timing performance in some cases because of the optimization and resource effects described above, causing longer logic delays. Floorplan assignments restrict logic placement, which can make it more difficult for the Fitter to meet timing requirements. Use the guidelines in this manual to reduce any effect on your design performance.

Related Information

- Design Partition Guidelines on page 14-9
- Checking Floorplan Quality on page 14-44

Turning On Supported Cross-Boundary Optimizations

You can improve the optimizations performed between design partitions by turning on the cross-boundary optimizations feature. You can select the optimizations as individual assignments for each partition. This allows the cross-boundary optimization feature to give you more control over the optimizations that work best for your design.

You can turn on the cross-boundary optimizations for your design partitions on the **Advanced** tab of the **Design Partition Properties** dialog box. Once you change the optimization settings, the Quartus II software recompiles your partition from source automatically. Cross-boundary optimizations include the following: propagate constants, propagate inversions on partition inputs, merge inputs fed by a common source, merge electrically equivalent bidirectional pins, absorb internal paths, and remove logic connected to dangling outputs.

Cross-boundary optimizations are implemented top-down from the parent partition into the child partition, but not vice-versa. The cross-boundary optimization feature cannot be used with partitions with multiple personas (partial reconfiguration partitions).

Although more partitions allow for a greater reduction in compilation time, consider limiting the number of partitions to prevent degradation in the quality of results. Creating good design partitions and good floorplan location assignments helps to improve the design resource utilization and timing performance results for cross-partition paths.

Related Information

Design Partition Properties Dialog Box online help

Guidelines for Incremental Compilation

General Partitioning Guidelines

The first step in planning your design partitions is to organize your source code so that it supports good partition assignments. Although you can assign any hierarchical block of your design as a design partition or merge hierarchical blocks into the same partition, following the design guidelines presented below ensures better results.

Plan Design Hierarchy and Design Files

You begin the partitioning process by planning the design hierarchy. When you assign a hierarchical instance as a design partition, the partition includes the assigned instance and entities instantiated below that are not defined as separate partitions. You can use the **Merge** command in the Design Partitions window to combine hierarchical partitions into a single partition, as long as they have the same immediate parent partition.

- When planning your design hierarchy, keep logic in the "leaves" of the hierarchy instead of having logic at the top-level of the design so that you can isolate partitions if required.
- Create entities that can form partitions of approximately equal size. For example, do not instantiate small entities at the same hierarchy level, because it is more difficult to group them to form reasonably-sized partitions.
- Create each entity in an independent file. The Quartus II software uses a file checksum to detect changes, and automatically recompiles a partition if its source file changes and its netlist type is set to either post-synthesis or post-fit. If the design entities for two partitions are defined in the same file, changes to the logic in one partition initiates recompilation for both partitions.
- Design dependencies also affect which partitions are compiled when a source file changes. If two partitions rely on the same lower-level entity definition, changes in that lower-level entity affect both partitions. Commands such as VHDL `use` and Verilog HDL `include` create dependencies between files, so that changes to one file can trigger recompilations in all dependent files. Avoid these types of file dependencies if possible. The Partition Dependent Files report for each partition in the Analysis & Synthesis section of the Compilation report lists which files contribute to each partition.

Using Partitions with Third-Party Synthesis Tools

Incremental compilation works well with third-party synthesis tools in addition to Quartus II Integrated Synthesis. If you use a third-party synthesis tool, set up your tool to create a separate Verilog Quartus Mapping File (**.vqm**) or EDIF Input File (**.edf**) netlist for each hierarchical partition. In the Quartus II software, designate the top-level entity from each netlist as a design partition. The **.vqm** or **.edf** netlist file is treated as the source file for the partition in the Quartus II software.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation** on page 3-1

Partition Design by Functionality and Block Size

Initially, you should partition your design along functional boundaries. In a top-level system block diagram, each block is often a natural design partition. Typically, each block of a system is relatively independent and has more signal interaction internally than interaction between blocks, which helps reduce optimizations between partition boundaries. Keeping functional blocks together means that synthesis and fitting can optimize related logic as a whole, which can lead to improved optimization.

- Consider how many partitions you want to maintain in your design to determine the size of each partition. Your compilation time reduction goal is also a factor, because compiling small partitions is typically faster than compiling large partitions.
- There is no minimum size for partitions; however, having too many partitions can reduce the quality of results by limiting optimization. Ensure that the design partitions are not too small. As a general guideline, each partition should contain more than approximately 2,000 logic elements (LEs) or adaptive logic modules (ALMs). If your design is incomplete when you partition the design, use previous designs to help estimate the size of each block.

Partition Design by Clock Domain and Timing Criticality

Consider which clock in your design feeds the logic in each partition. If possible, keep clock domains within one partition. When a clock signal is isolated to one partition, it reduces dependence on other partitions for timing optimization. Isolating a clock domain to one partition also allows better use of regional clock routing networks if the partition logic is constrained to one region of the design. Additionally, limiting the number of clocks within each partition simplifies the timing requirements for each partition during optimization. Use an appropriate subsystem to implement the required logic for any clock domain transfers (such as a synchronization circuit, dual-port RAM, or FIFO). You can include this logic inside the partition at one side of the transfer.

Try to isolate timing-critical logic from logic that you expect to easily meet timing requirements. Doing so allows you to preserve the satisfactory results for non-critical partitions and focus optimization iterations on only the timing-critical portions of the design to minimize compilation time.

Related Information

Analyzing and Optimizing the Design Floorplan with the Chip Planner documentation
Information about clock domains and their affect on partition design

Consider What Is Changing

When assigning partitions, you should consider what is changing in the design. Is there intellectual property (IP) or reused logic for which the source code will not change during future design iterations? If so, define the logic in its own partition so that you can compile one time and immediately preserve the results and not have to compile that part of the design again. Is logic being tuned or optimized, or are specifications changing for part of the design? If so, define changing logic in its own partition so that you can recompile only the changing part while the rest of the design remains unchanged.

As a general rule, create partitions to isolate logic that will change from logic that will not change. Partitioning a design in this way maximizes the preservation of unchanged logic and minimizes compilation time.

Design Partition Guidelines

Follow the design partition guidelines below when you create or modify the HDL code for each design block that you might want to assign as a design partition. You do not need to follow all the recommendations exactly to achieve a good quality of results with the incremental compilation flow, but adhering to as many as possible maximizes your chances for success.

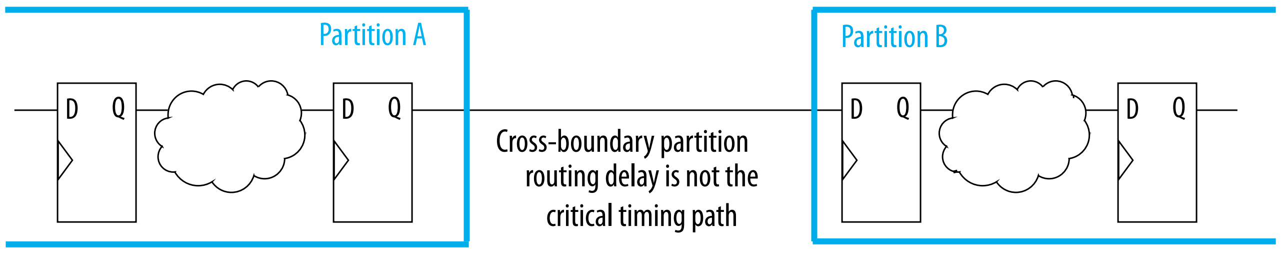
The design partition guidelines include examples of the types of optimizations that are prevented by partition boundaries, and describes how you can structure or modify your partitions to avoid these limitations.

Register Partition Inputs and Outputs

Use registers at partition input and output connections that are potentially timing-critical. Registers minimize the delays on inter-partition paths and prevent the need for cross-boundary optimizations.

If every partition boundary has a register as shown in the figure, every register-to-register timing path between partitions includes only routing delay. Therefore, the timing paths between partitions are likely not timing-critical, and the Fitter can generally place each partition independently from other partitions. This advantage makes it easier to create floorplan location assignments for each separate partition, and is especially important for flows in which partitions are placed independently in separate Quartus II projects. Additionally, the partition boundary does not affect combinational logic optimization because each register-to-register logic path is contained within a single partition.

Figure 14-2: Registering Partition I/O


Partition A
D Q
D Q
Cross-boundary partition routing delay is not the critical timing path
Partition B
D Q
D Q

If a design cannot include both input and output registers for each partition due to latency or resource utilization concerns, choose to register one end of each connection. If you register every partition output, for example, the combinational logic that occurs in each cross-partition path is included in one partition so that it can be optimized together.

It is a good synchronous design practice to include registers for every output of a design block. Registered outputs ensure that the input timing performance for each design block is controlled exclusively within the destination logic block.

Related Information

- **Partition Statistics Report** on page 14-31
- **Incremental Compilation Advisor** on page 14-28

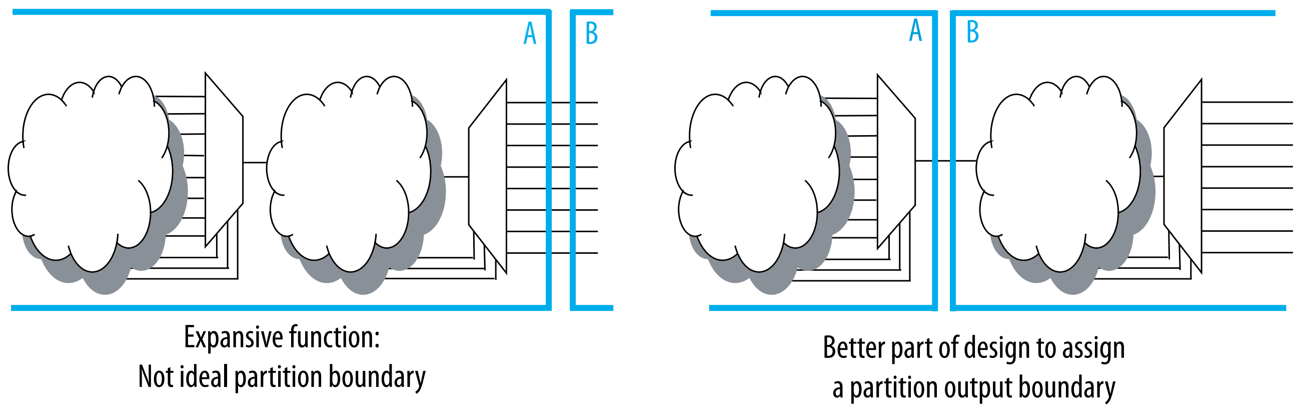
Minimize Cross-Partition-Boundary I/O

Minimize the number of I/O paths that cross between partition boundaries to keep logic paths within a single partition for optimization. Doing so makes partitions more independent for both logic and placement optimization.

This guideline is most important for timing-critical and high-speed connections between partitions, especially in cases where the input and output of each partition is not registered. Slow connections that are not timing-critical are acceptable because they should not impact the overall timing performance of the design. If there are timing-critical paths between partitions, rework or merge the partitions to avoid these inter-partition paths.

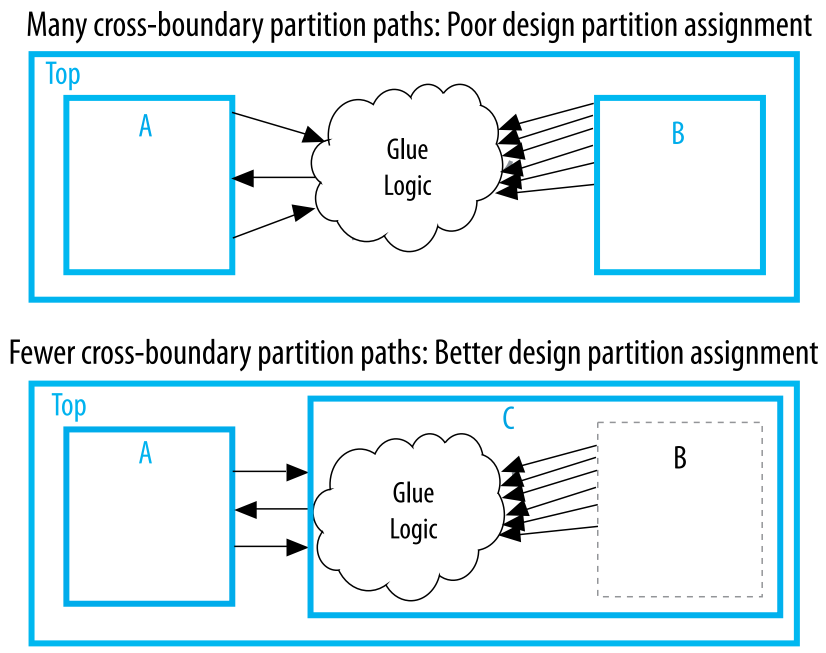
When dividing your design into partitions, consider the types of functions at the partition boundaries. The figure shows an expansive function with more outputs than inputs in the left diagram, which makes a poor partition boundary, and, on the right side, a better place to assign the partition boundary that minimizes
cross-partition I/Os. Adding registers to one or both sides of the cross-partition path in this example would further improve partition quality.

Figure 14-3: Minimizing I/O Between Partitions by Moving the Partition Boundary


A
B
Expansive function:
Not ideal partition boundary
A
B
Better part of design to assign
a partition output boundary

Another way to minimize connections between partitions is to avoid using combinational ”glue logic” between partitions. You can often move the logic to the partition at one end of the connection to keep more logic paths within one partition. For example, the bottom diagram includes a new level of hierarchy C defined as a partition instead of block B. Clearly, there are fewer I/O connections between partitions A and C than between partitions A and B.

Figure 14-4: Minimizing I/O between Partitions by Modifying Glue Logic


Many cross-boundary partition paths: Poor design partition assignment
Top
A
Glue Logic
B
Fewer cross-boundary partition paths: Better design partition assignment
Top
A
C
Glue Logic
B

Related Information

- **Partition Statistics Report** on page 14-31
- **Incremental Compilation Advisor** on page 14-28

Examine the Need for Logic Optimization Across Partitions

Partition boundaries prevent logic optimizations across partitions (except for some limited cross-boundary optimizations).

In some cases, especially if part of the design is complete or comes from another designer, the designer might not have followed these guidelines when the source code was created. These guidelines are not mandatory to implement an incremental compilation flow, but can improve the quality of results. If assigning a partition affects resource utilization or timing performance of a design block as compared to the flat design, it might be due to one of the issues described in the logic optimization across partitions guidelines below. Many of the examples suggest simple changes to your partition definitions or hierarchy to move the partition boundary to improve your results.

The following guidelines ensure that your design does not require logic optimization across partition boundaries:

Keep Logic in the Same Partition for Optimization and Merging

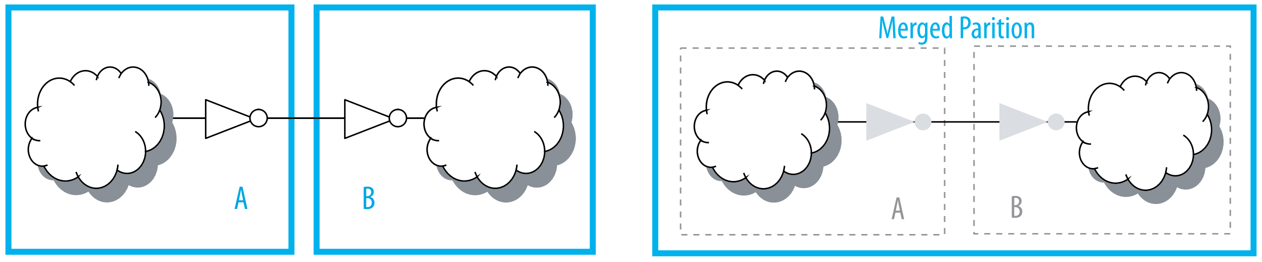
If your design logic requires logic optimization or merging to obtain optimal results, ensure that all the logic is part of the same partition because only limited cross-boundary optimizations are permitted.

Example—Combinational Logic Path

If a combinational logic path is split across two partitions, the logic cannot be optimized or merged into one logic cell in the device. This effect can result in an extra logic cell in the path, increasing the logic delay. As a very simple example, consider two inverters on the same signal in two different partitions, A and B, as shown in the left diagram of the figure. To maintain correct incremental functionality, these two inverters cannot be removed from the design during optimization because they occur in different design partitions. The Quartus II software cannot use information about other partitions when it compiles each partition, because each partition is allowed to change independently from the other.

On the right side of the figure, partitions A and B are merged to group the logic in blocks A and B into one partition. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchy block as the partition. With the logic contained in one partition, the software can optimize the logic and remove the two inverters (shown in gray), which reduces the delay for that logic path. Removing two inverters is not a significant reduction in resource utilization because inversion logic is readily available in Altera device architecture. However, this example is a simple demonstration of the types of logic optimization that are prevented by partition boundaries.

Figure 14-5: Keeping Logic in the Same Partition for Optimization



Inverters in separate partitions A and B cannot be removed from design: Poor design partition assignment

Inverters in merged partition can be removed: Better design partition assignment

Example—Fitter Merging

In a flat design, the Fitter can also merge logical instantiations into the same physical device resource. With incremental compilation, logic defined in different partitions cannot be merged to use the same physical device resource.

For example, the Fitter can merge two single-port RAMs from a design into one dedicated RAM block in the device. If the two RAMs are defined in different partitions, the Fitter cannot merge them into one dedicated device RAM block.

This limitation is a only a concern if merging is required to fit the design in the target device. Therefore, you are more likely to encounter this issue during troubleshooting rather than during planning, if your design uses more logic than is available in the device.

Merging PLLs and Transceivers (GXB)

Multiple instances of the ALTPLL IP core can use the same PLL resource on the device. Similarly, GXB transceiver instances can share high-speed serial interface (HSSI) resources in the same quad as other instances. The Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions. Therefore, there are no restrictions for PLLs and high-speed transceiver blocks when setting up partitions.

Keep Constants in the Same Partition as Logic

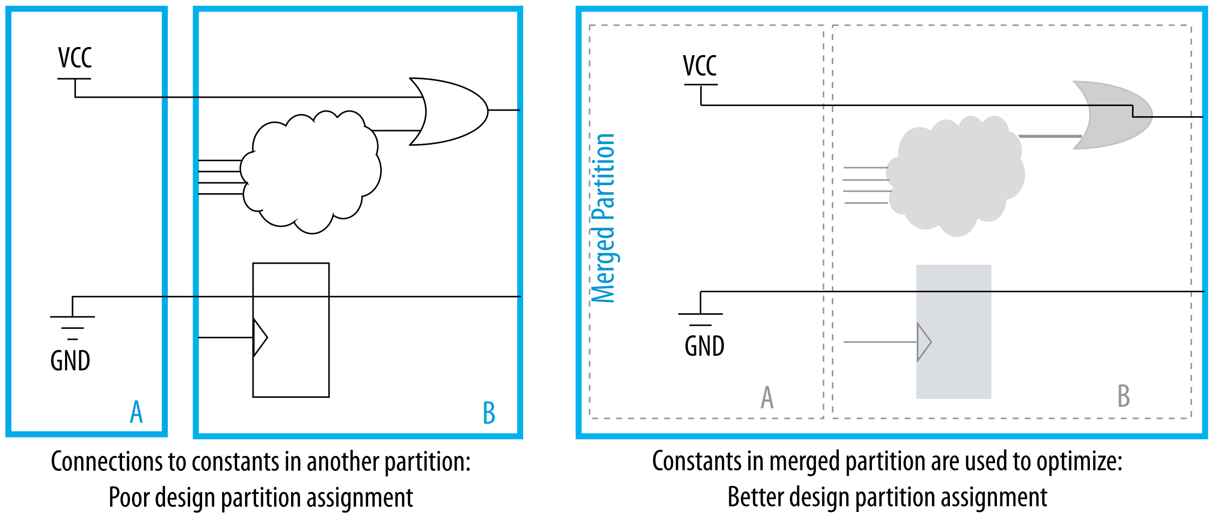
Because the Quartus II software cannot fully optimize across a partition boundary, constants are not propagated across partition boundaries, except from parent partition to child partition. A signal that is constant (1/V_{CC} or 0/GND) in one partition cannot affect another partition.

Example—Constants in Merged Partitions

For example, the left diagram of the figure shows part of a design in which partition A defines some signals as constants (and assumes that the other input connections come from elsewhere in the design and are not shown in the figure). Constants such as these could appear due to parameter or generic settings or configurations with parameters, setting a bus to a specific set of values, or could result from optimizations that occur within a group of logic. Because the blocks are independent, the software cannot optimize the logic in block B based on the information from block A. The right side of the figure shows a merged partition that groups the logic in blocks A and B. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchical block as the partition.

Within the single merged partition, the Quartus II software can use the constants to optimize and remove much of the logic in block B (shown in gray), as shown in the figure.

Figure 14-6: Keeping Constants in the Same Partition as the Logic They Feed


VCC
GND
A
B
Connections to constants in another partition:
Poor design partition assignment
Merged Partition
VCC
GND
A
B
Constants in merged partition are used to optimize:
Better design partition assignment

Related Information

- **Partition Statistics Report** on page 14-31
- **Incremental Compilation Advisor** on page 14-28

Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together

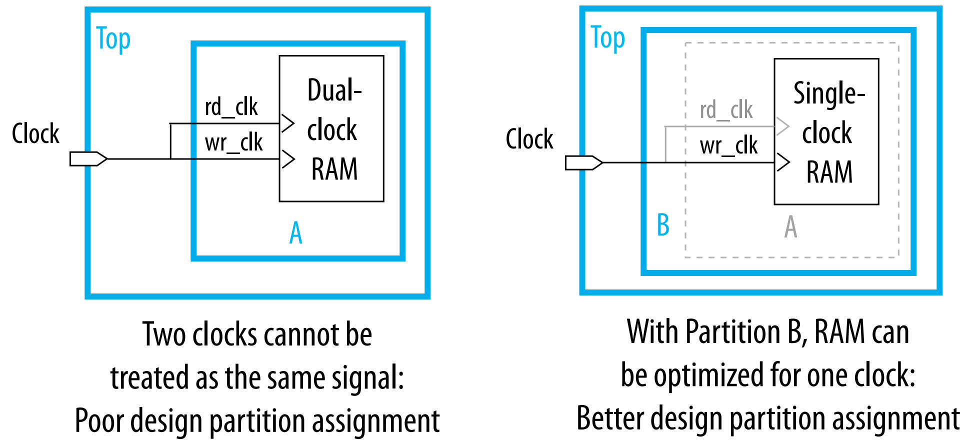
Do not use the same signal to drive multiple ports of a single partition or directly connect two ports of a partition. If the same signal drives multiple ports of a partition, or if two ports of a partition are directly connected, those ports are logically equivalent. However, the software has limited information about connections made in another partition (including the top-level partition), the compilation cannot take advantage of the equivalence. This restriction usually produces sub-optimal results.

If your design has these types of connections, redefine the partition boundaries to remove the affected ports. If one signal from a higher-level partition feeds two input ports of the same partition, feed the one signal into the partition, and then make the two connections within the partition. If an output port drives an input port of the same partition, the connection can be made internally without going through any I/O ports. If an input port drives an output port directly, the connection can likely be implemented without the ports in the lower-level partition by connecting the signals in a higher-level partition.

Example—Single Signal Driving More Than One Port

The figure shows an example of one signal driving more than one port. The left diagram shows a design where a single clock signal is used to drive both the read and write clocks of a RAM block. Because the RAM block is compiled as a separate partition A, the RAM block is implemented as though there are two unique clocks. If you know that the port connectivity will not change (that is, the ports will always be driven by the same signal in the top-level partition), redefine the port interface so that there is only a single port that can drive both connections inside the partition. You can create a wrapper file to define a partition that has fewer ports, as shown in the diagram on the right side. With the single clock fed into the partition, the RAM can be optimized into a single-clock RAM instead of a dual-clock RAM. Single-clock RAM can provide better performance in the device architecture. Additionally, partition A might use two global routing lines for the two copies of the clock signal. Partition B can use one global line that fans out to all destinations. Using just the single port connection prevents overuse of global routing resources.

Figure 14-7: Preventing One Signal from Driving Multiple Partition Inputs


Top
Clock
rd_clk
wr_clk
Dual-clock RAM
A
Two clocks cannot be treated as the same signal: Poor design partition assignment
Top
Clock
rd_clk
wr_clk
Single-clock RAM
B
A
With Partition B, RAM can be optimized for one clock: Better design partition assignment

Related Information

Incremental Compilation Advisor on page 14-28

Invert Clocks in Destination Partitions

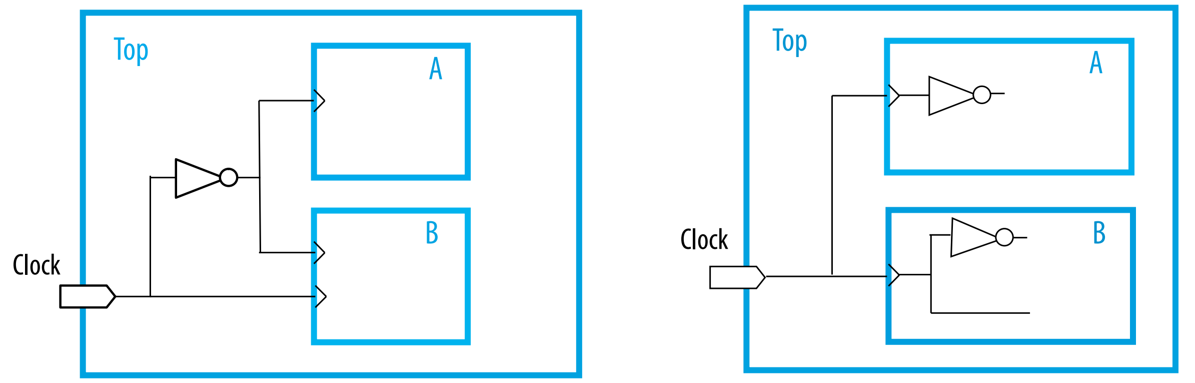
For best results, clock inversion should be performed in the destination logic array block (LAB) because each LAB contains clock inversion circuitry in the device architecture. In a flat compilation, the Quartus II software can optimize a clock inversion to propagate it to the destination LABs regardless of where the inversion takes place in the design hierarchy. However, clock inversion cannot propagate through a partition boundary (except from a parent partition to a child partition) to take advantage of the inversion architecture in the destination LABs.

Example—Clock Signal Inversion

With partition boundaries as shown in the left diagram of the figure, the Quartus II software uses logic to invert the signal in the partition that defines the inversion (the top-level partition in this example), and then routes the signal on a global clock resource to its destinations (in partitions A and B). The inverted clock acts as a gated clock with high skew. A better solution is to invert the clock signal in the destination partitions as shown on the right side of the diagram. In this case, the correct logic and routing resources can be used, and the signal does not behave like a gated clock.

The figure shows the clock signal inversion in the destination partitions.

Figure 14-8: Inverting Clock Signal in Destination Partitions



Inverter acts as clock gating (adding skew):
Poor design partition assignment

Clock inverted inside destination LABs,
only one global routing signal:
Better design partition assignment

Notice that this diagram also shows another example of a single pin feeding two ports of a partition boundary. In the left diagram, partition B does not have the information that the clock and inverted clock come from the same source. In the right diagram, partition B has more information to help optimize the design because the clock is connected as one port of the partition.

Connect I/O Pin Directly to I/O Register for Packing Across Partition Boundaries

The Quartus II software allows cross-partition register packing of I/O registers in certain cases where your input and output pins are defined in the top-level hierarchy (and the top-level partition), but the corresponding I/O registers are defined in other partitions.

Input pin cross-partition register packing requires the following specific circumstances:

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

Output pin cross-partition register packing requires the following specific circumstances:

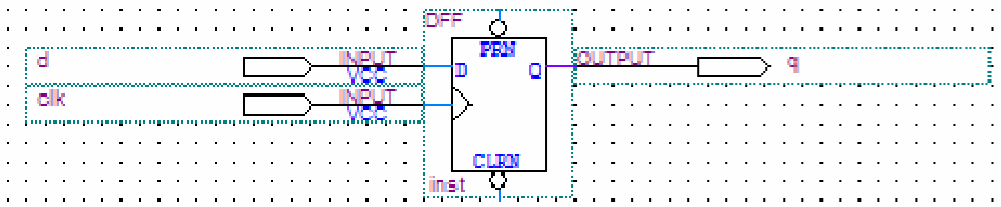
- The register feeds exactly one output pin.
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

The following examples of I/O register packing illustrate this point using Block Design File (**.bdf**) schematics to describe the design logic.

Example 1—Output Register in Partition Feeding Multiple Output Pins

In this example, the subdesign contains a single register.

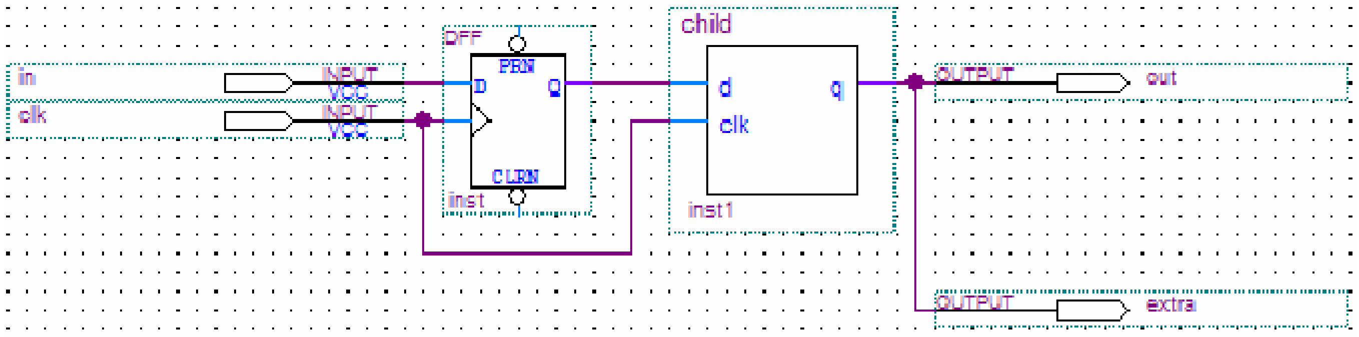
Figure 14-9: Subdesign with One Register, Designated as a Separate Partition



If the top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, and designates the subdesign as a separate design partition, the Quartus II software can perform cross-partition register packing because the single partition port feeds the output pin directly.

In this example, the top-level design instantiates the subdesign as an output register with more than one fan-out signal.

Figure 14-10: Top-Level Design Instantiating the Subdesign with Two Output Pins



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

This type of cross-partition register packing is not allowed because it requires modification to the interface of the subdesign partition. To perform incremental compilation, the Quartus II software must preserve the interface of design partitions.

To allow the Quartus II software to pack the register in the subdesign with the output pin `out` in the figure, restructure your HDL code so that output registers directly connect to output pins by making one of the following changes:

- Place the register in the same partition as the output pin. The simplest method is to move the register from the subdesign partition into the partition containing the output pin. Doing so guarantees that the Fitter can optimize the two nodes without violating partition boundaries.
- Duplicate the register in your subdesign HDL so that each register feeds only one pin, and then connect the extra output pin to the new port in the top-level design. Doing so converts the cross-partition register packing into the simplest case where each register has a single fan-out.

Figure 14-11: Modified Subdesign with Two Output Registers and Two Output Ports

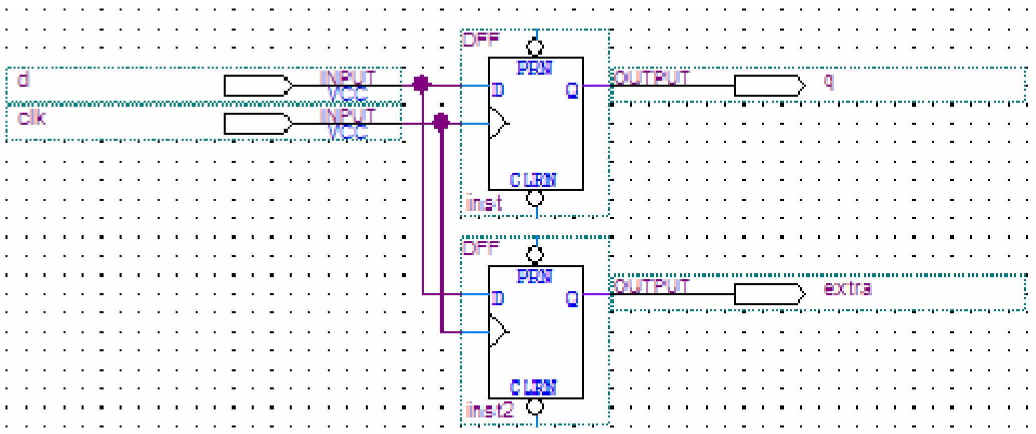
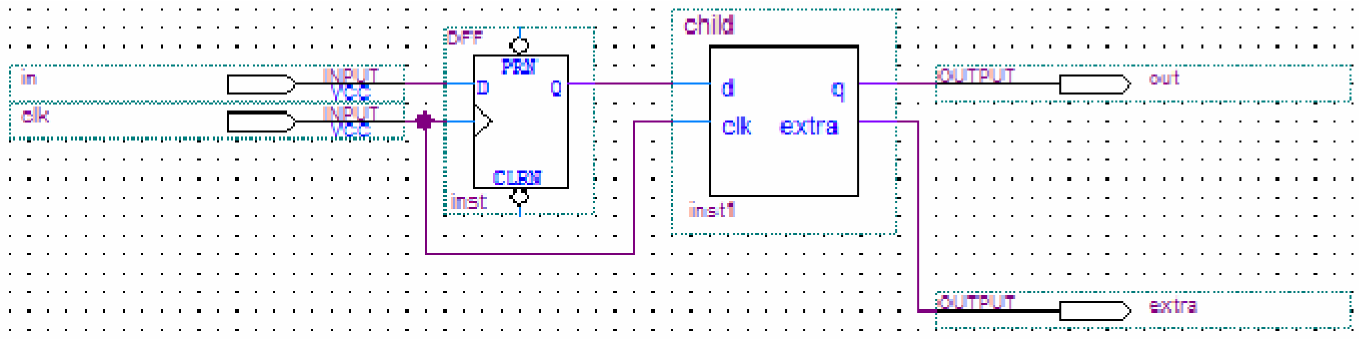


Figure 14-12: Modified Top-Level Design Connecting Two Output Ports to Output Pins



Example 2—Input Register in Partition Fed by an Inverted Input Pin or Output Register in Partition Feeding an Inverted Output Pin

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

Figure 14-13: Top-Level Design Instantiating Subdesign as an Input Register with an Inverted Input Pin

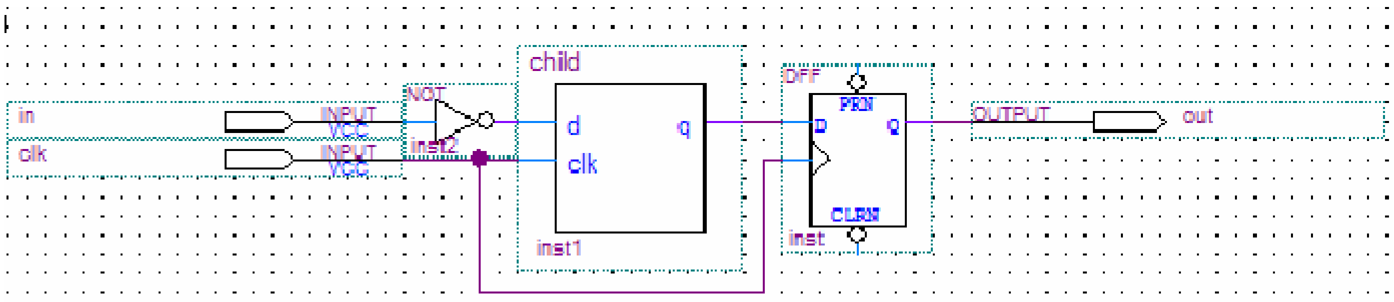
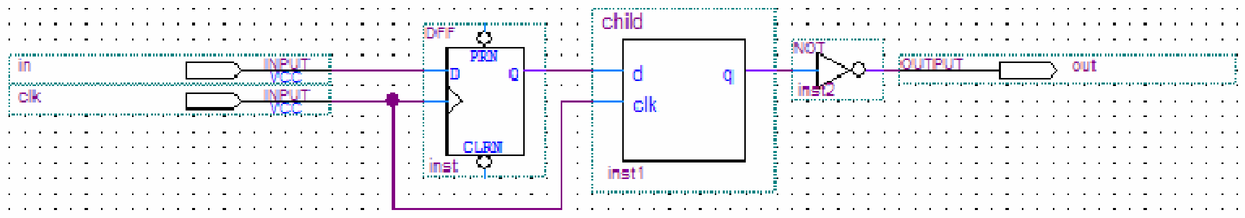


Figure 14-14: Top-Level Design Instantiating the Subdesign as an Output Register Feeding an Inverted Output Pin



In these cases, the Quartus II software does not perform register packing. If there is a **Fast Input Register** assignment on pin `in,` as shown in the top figure, or a **Fast Output Register** assignment on pin `out,` as shown in the bottom figure, the Quartus II software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and I/O cell are connected across a design partition boundary.

This type of register packing is not allowed because it requires moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the register must be moved out of the subdesign partition, or the inverter must be moved into the subdesign partition to be implemented in the register.

To allow the Quartus II software to pack the single register in the subdesign with the input pin `in`, as shown in top figure or the output pin `out`, as shown in the bottom figure, restructure your HDL code to place the register in the same partition as the inverter by making one of the following changes:

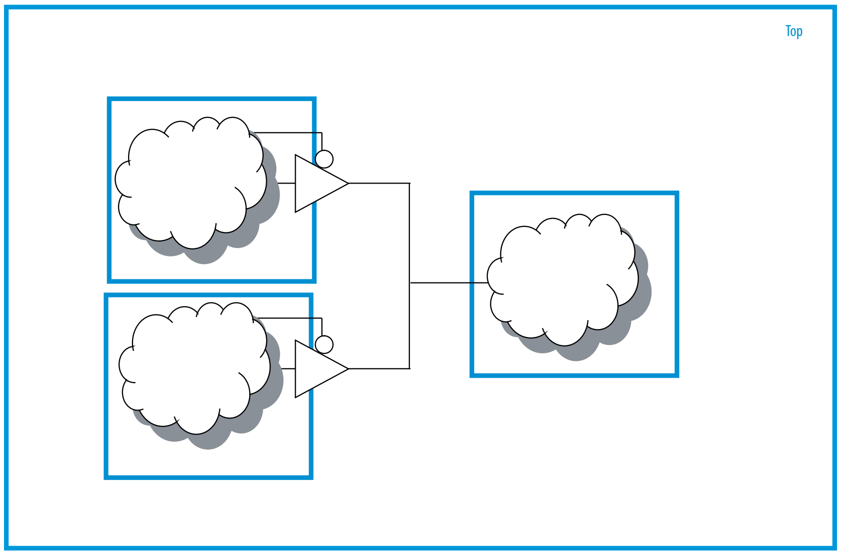
- Move the register from the subdesign partition into the top-level partition containing the pin. Doing so ensures that the Fitter can optimize the I/O register and inverter without violating partition boundaries.
- Move the inverter from the top-level block into the subdesign, and then connect the subdesign directly to a pin in the top-level design. Doing so allows the Fitter to optimize the inverter into the register implementation, so that the register is directly connected to a pin, which enables register packing.

Do Not Use Internal Tri-States

Internal tri-state signals are not recommended for FPGAs because the device architecture does not include internal tri-state logic. If designs use internal tri-states in a flat design, the tri-state logic is usually converted to `OR` gates or multiplexing logic. If tri-state logic occurs on a hierarchical partition boundary, the Quartus II software cannot convert the logic to combinational gates because the partition could be connected to a top-level device I/O through another partition.

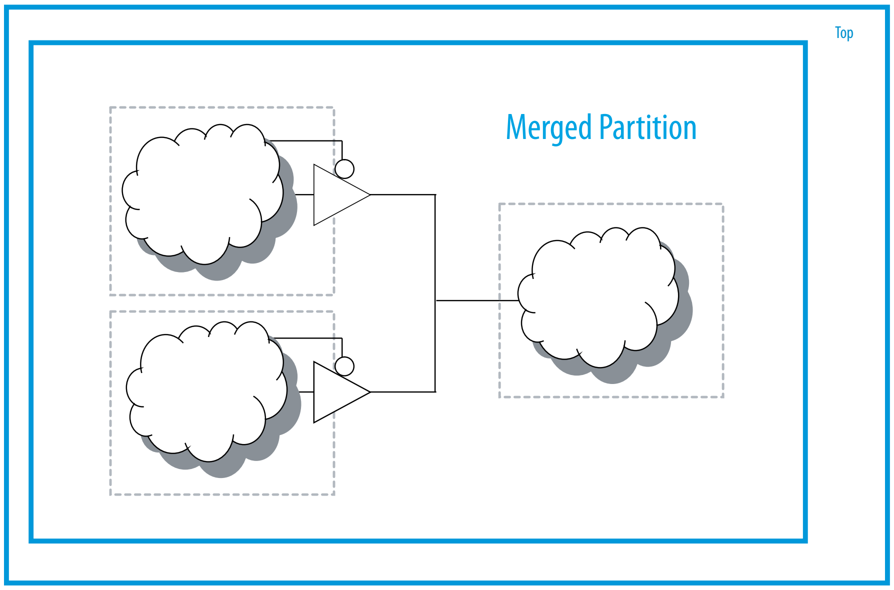
The figures below show a design with partitions that are not supported for incremental compilation due to the internal tri-state output logic on the partition boundaries. Instead of using internal tri-state logic for partition outputs, implement the correct logic to select between the two signals. Doing so is good practice even when there are no partitions, because such logic explicitly defines the behavior for the internal signals instead of relying on the Quartus II software to convert the tri-state signals into logic.

Figure 14-15: Unsupported Internal Tri-State Signals



Design results in Quartus II error message:
The software cannot synthesize this
design and maintain incremental functionality.

Figure 14-16: Merged Partition Allows Synthesis to Convert Internal Tri-State Logic to Combinational Logic



Merged partition allows synthesis to
convert tri-state logic into
combinational logic.

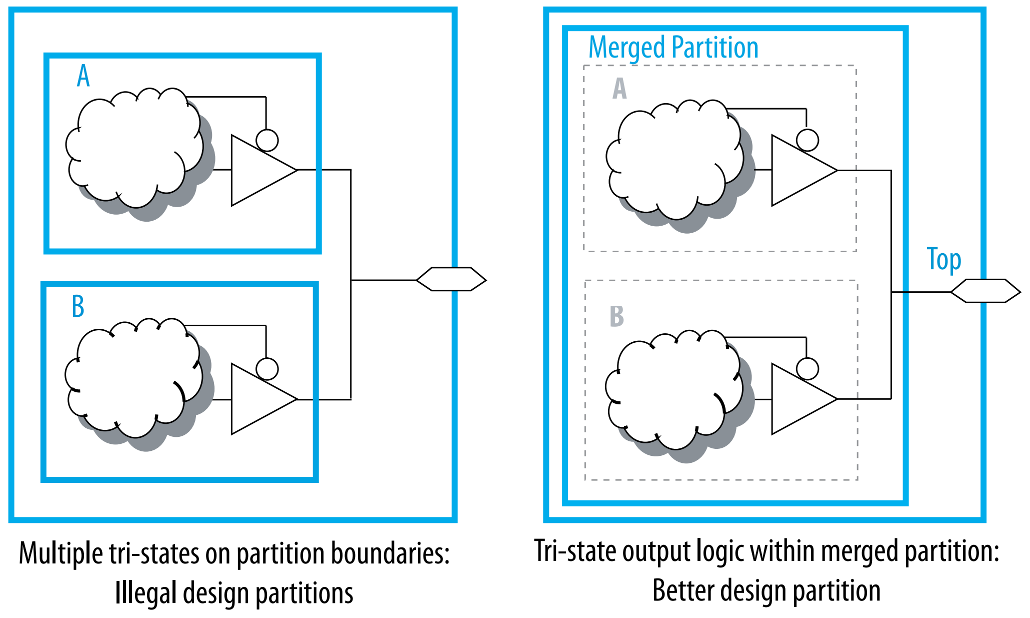
Do not use tri-state signals or bidirectional ports on hierarchical partition boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you must use internal tri-state logic, ensure that all the control and destination logic is contained in the same partition, in which case the Quartus II software can convert the internal tri-state signals into combinational logic as in a flat design. In this example, you can also merge all three partitions into one partition, as shown in the bottom figure, to allow the Quartus II software to treat the logic as internal tri-state and perform the same type of optimization as a flat design. If possible, you should avoid using internal tri-state logic in any Altera FPGA design to ensure that you get the desired implementation when the design is compiled for the target device architecture.

Include All Tri-State and Enable Logic in the Same Partition

When multiple output signals use tri-state logic to drive a device output pin, the Quartus II software merges the logic into one tri-state output pin. The Quartus II software cannot merge tri-state outputs into one output pin if any of the tri-state logic occurs on a partition boundary. Similarly, output pins with an output enable signal cannot be packed into the device I/O cell if the output enable logic is part of a different partition from the output register. To allow register packing for output pins with an output enable signal, structure your HDL code or design partition assignments so that the register and enable logic are defined in the same partition.

The figure shows a design with tri-state output signals that feed a device bidirectional I/O pin (assuming that the input connection feeds elsewhere in the design and is not shown in the figure). In the left diagram below, the tri-state output signals appear as the outputs of two separate partitions. In this case, the Quartus II software cannot implement the specified logic and maintain incremental functionality. In the right diagram, partitions A and B are merged to group the logic from the two blocks. With this single partition, the Quartus II software can merge the two tri-state output signals and implement them in the tri-state logic available in the device I/O element.

Figure 14-17: Including All Tri-State Output Logic in the Same Partition


A
B
Merged Partition
A
B
Top
Multiple tri-states on partition boundaries: Illegal design partitions
Tri-state output logic within merged partition: Better design partition

Summary of Guidelines Related to Logic Optimization Across Partitions

To ensure that your design does not require logic optimization across partitions, follow the guidelines below:

- Include logic in the same partition for optimization and merging
- Include constants in the same partition as logic
- Avoid signals that drive multiple partition I/O or connect I/O together
- Invert clocks in destination partitions

- Connect I/O directly to I/O register for packing across partition boundaries
- Do not use internal tri-states
- Include all tri-state and enable logic in the same partition

Remember that these guidelines are not mandatory when implementing an incremental compilation flow, but can improve the quality of results. When creating source design code, follow these guidelines and organize your HDL code to support good partition boundaries. For designs that are complete, assess whether assigning a partition affects the resource utilization or timing performance of a design block as compared to the flat design. Make the appropriate changes to your design or hierarchy, or merge partitions as required, to improve your results.

Consider a Cascaded Reset Structure

Designs typically have a global asynchronous reset signal where a top-level signal feeds all partitions. To minimize skew for the high fan-out signal, the global reset signal is typically placed onto a global routing resource.

In some cases, having one global reset signal can lead to recovery and removal time problems. This issue is not specific to incremental flows; it could be applicable in any large high-speed design. In an incremental flow, the global reset signal creates a timing dependency between the top-level partition and lower-level partitions.

For incremental compilation, it is helpful to minimize the impact of global structures. To isolate each partition, consider adding reset synchronizers. Using cascaded reset structures, the intent is to reduce the inter-partition fan-out of the reset signal, thereby minimizing the effect of the global signal. Reducing the fan-out of the global reset signal also provides more flexibility in routing the cascaded signals, and might help recovery and removal times in some cases.

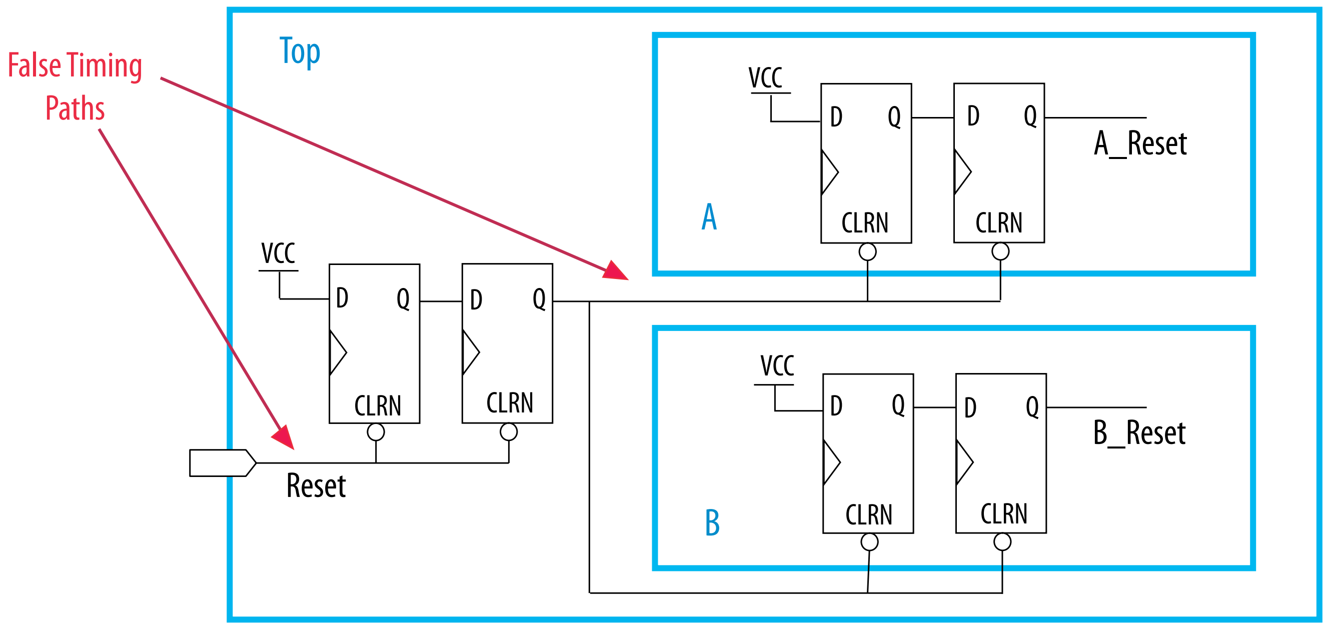
This recommendation can help in large designs, regardless of whether you are using incremental compilation. However, if one global signal can feed all the logic in its domain and meet recovery and removal times, this recommendation may not be applicable for your design. Minimizing global structures is more relevant for
high-performance designs where meeting timing on the reset logic can be challenging. Isolating each partition and allowing more flexibility in global routing structures is an additional advantage in incremental flows.

If you add additional reset synchronizers to your design, latency is also added to the reset path, so ensure that this is acceptable in your design. Additionally, parts of the design may come out of the reset state in different clock cycles. You can balance the latency or add hand-shaking logic between partitions, if necessary, to accommodate these differences.

The signal is first synchronized on the chip following good synchronous design practices, meaning that the design asynchronously resets, but synchronously releases from reset to avoid any race conditions or metastability problems. Then, to minimize the impact of global structures, the circuit employs a divide-and-conquer approach for the reset structure. By implementing a cascaded reset structure, the reset paths for each partition are independent. This structure reduces the effect of inter-partition dependency because the inter-partition reset signals can now be treated as false paths for timing analysis. In some cases, the reset signal of the partition can be placed on local lines to reduce the delay added by routing to a global routing line. In other cases, the signal can be routed on a regional or quadrant clock signal.

The figure shows a cascaded reset structure.

Figure 14-18: Cascaded Reset Structure



This circuit design can help you achieve timing closure and partition independence for your global reset signal. Evaluate the circuit and consider how it works for your design.

Related Information

- **Recommended Design Practices documentation** on page 11-1
  Information and design recommendations for reset structures

Design Partition Guidelines for Third-Party IP Delivery

There are additional design guidelines that can improve incremental compilation flows where exported partitions are developed independently. These guidelines are not always required, but are usually recommended if the design includes partitions compiled in a separate Quartus II project, such as when delivering intellectual property (IP). A unique challenge of IP delivery for FPGAs is the fact that the partitions developed independently must share a common set of resources. To minimize issues that might arise from sharing a common set of resources, you can design partitions within a single Quartus II project, or a copy of the top-level design. A common project ensures that designers have a consistent view of the top-level design framework.

Alternatively, an IP designer can export just the post-synthesis results to be integrated in the top-level design when the post-fitting results from the IP project are not required. Using a post-synthesis netlist provides more flexibility to the Quartus II Fitter, so that less resource allocation is required. If a common project is not possible, especially when the project lead plans to integrate the IP's post-fitting results, it is important that the project lead and IP designer clearly communicate their requirements.

Related Information

Project Management in Team-Based Design Flows on page 14-4

Allocate Logic Resources

In an incremental compilation design flow in which designers, such as third-party IP providers, optimize partitions and then export them to a top-level design, the Quartus II software places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level. Allocation of logic resources requires that you decide on a set of logic resources (including I/O, LAB logic blocks, RAM and DSP blocks) that the IP block will ”own”. This process can be interactive; the project lead and the IP designer might work together to determine what resources are required for the IP block and are available in the top-level design.

You can constrain logic utilization for the IP core using design floorplan location assignments. The design should specify I/O pin locations with pin assignments.

You can also specify limits for Quartus II synthesis to allocate and balance resources. This procedure can also help if device resources are overused in the individual partitions during synthesis.

In the standard synthesis flow, the Quartus II software can perform automated resource balancing for DSP blocks or RAM blocks and convert some of the logic into regular logic cells to prevent overuse.

You can use the Quartus II synthesis options to control inference of IP cores that use the DSP, or RAM blocks. You can also use the IP Catalog and Parameter Editor to customize your RAM or DSP IP cores to use regular logic instead of the dedicated hardware blocks.

Related Information

- **Introduction to Design Floorplans** on page 14-36
- **Quartus II Integrated Synthesis documentation** on page 16-1
  Information about resource balancing DSP and RAM blocks when using Quartus II synthesis
- **Timing Closure and Optimization documentation**
  Tips about resource balancing and reducing resource utilization
- **More Analysis Synthesis Settings Dialog Box online help**
  Information about how to set global logic options for partitions

Allocate Global Routing Signals and Clock Networks if Required

In most cases, you do not have to allocate global routing signals because the Quartus II software finds the best solution for the global signals. However, if your design is complex and has multiple clocks, especially for a partition developed by a third-party IP designer, you may have to allocate global routing resources between various partitions.

Global routing signals can cause conflicts when independent partitions are integrated into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Third-party partitions can use the same global routing resources, thus causing conflicts in the top-level design. Additionally, LAB placement depends on whether the inputs to the logic cells within the LAB use a global clock signal. Problems can occur if a design does not use a global signal in a lower-level partition, but does use a global signal in the top-level design.

If the exported IP core is small, you can reduce the potential for problems by using constraints to promote clock and high fan-out signals to regional routing signals that cover only part of the device, instead of global routing signals. In this case, the Quartus II software is likely to find a routing solution in the top-level design because there are many regional routing signals available on most Altera devices, and designs do not typically overuse regional resources.

To ensure that an IP block can utilize a regional clock signal, view the resource coverage of regional clocks in the Chip Planner, and then align LogicLock regions that constrain partition placement with available global clock routing resources. For example, if the LogicLock region for a particular partition is limited to one device quadrant, that partition's clock can use a regional clock routing type that covers only one device quadrant. When all partition logic is available, the project lead can compile the entire design at the top level with floorplan assignments to allow the use of regional clocks that span only a part of the device.

If global resources are heavily used in the overall design, or the IP designer requires global clocks for their partition, you can set up constraints to avoid signal overuse at the top-level by assigning the appropriate type of global signals or setting a maximum number of clock signals for the partition.

You can use the **Global Signal** assignment to force or prevent the use of a global routing line, making the assignment to a clock source node or signal. You can also assign certain types of global clock resources in some device families, such as regional clocks. For example, if you have an IP core, such as a memory interface that specifies the use of a dual regional clock, you can constrain the IP to part of the device covered by a regional clock and change the **Global Signal** assignment to use a regional clock. This type of assignment can reduce clocking congestion and conflicts.

Alternatively, partition designers can specify the number of clocks allowed in the project using the maximum clocks allowed options in the **Advanced Settings (Fitter)** dialog box. Specify **Maximum number of clocks of any type allowed**, or use the **Maximum number of global clocks allowed**, **Maximum number of regional clocks allowed**, and **Maximum number of periphery clocks allowed** options to restrict the number of clock resources of a particular type in your design.

If you require more control when planning a design with integrated partitions, you can assign a specific signal to use a particular clock network in newer device families by assigning the clock control block instance called CLKCTRL. You can make a point-to-point assignment from a clock source node to a destination node, or a single-point assignment to a clock source node with the **Global Clock CLKCTRL Location** logic option. Set the assignment value to the name of the clock control block: `CLKCTRL_G`<*global network number*> for a global routing network, or `CLKCTRL_R`<*regional network number*> for a dedicated regional routing network in the device.

If you want to disable the automatic global promotion performed in the Fitter to prevent other signals from being placed on global (or regional) routing networks, turn off the **Auto Global Clock** and **Auto Global Register Control Signals** options in the **Advanced Settings (Fitter)** dialog box.

If you are using design partition scripts for independent partitions, the Quartus II software can automatically write the commands to pass global constraints and turn off automatic options.

Alternatively, to avoid problems when integrating partitions into the top-level design, you can direct the Fitter to discard the placement and routing of the partition netlist by using the post-synthesis netlist, which forces the Fitter to reassign all the global signals for the partition when compiling the top-level design.

Related Information

- **Advanced Settings (Fitter) Dialog Box online help**
  Information about how to disable automatic global promotion
- **Generating Design Partition Scripts for Project Management online help**
- **Analyzing and Optimizing the Design Floorplan with the Chip Planner documentation**
  Information about how clock networks affect partition design

Assign Virtual Pins

Virtual pins map lower-level design I/Os to internal cells. If you are developing an IP block in an independent Quartus II project, use virtual pins when the number of I/Os on a partition exceeds the device I/O count, and to increase the timing accuracy of cross-partition paths.

You can create a virtual pin assignment in the Assignment Editor for partition I/Os that will become internal nodes in the top-level design. When you apply the Virtual Pin assignment to an input pin, the pin no longer appears as an FPGA pin, but is fixed to GND or VCC in the design. The assigned pin is not an open node. Leave the clock pins mapped to I/O pins to ensure proper routing.

You can specify locations for the virtual pins that correspond to the placement of other partitions, and also make timing assignments to the virtual pins to define a timing budget. Virtual pins are created automatically from the top-level design if you use design partition scripts. The scripts place the virtual pins to correspond with the placement of the other partitions in the top-level design.

Note: Tri-state outputs cannot be assigned as virtual pins because internal tri-state signals are not supported in Altera devices. Connect the signal in the design with regular logic, or allow the software to implement the signal as an external device I/O pin.

Related Information

Generating Design Partition Scripts for Project Management online help

Perform Timing Budgeting if Required

If you optimize partitions independently and integrate them to the top-level design, or compile with empty partitions, any unregistered paths that cross between partitions are not optimized as entire paths. In these cases, the Quartus II software has no information about the placement of the logic that connects to the I/O ports. If the logic in one partition is placed far away from logic in another partition, the routing delay between the logic can lead to problems in meeting timing requirements. You can reduce this effect by ensuring that input and output ports of the partitions are registered whenever possible. Additionally, using the same top-level project framework helps to avoid this problem by providing the software with full information about other design partitions in the top-level design.

To ensure that the software correctly optimizes the input and output logic in any independent partitions, you might be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Assigning a timing budget for each part of the connection ensures that the software optimizes the paths appropriately.

When performing manual timing budgeting in a partition for I/O ports that become internal partition connections in a top-level design, you can assign location and timing constraints to the virtual pin that represents each connection to further improve the quality of the timing budget.

Note: If you use design partition scripts, the Quartus II software can write I/O timing budget constraints automatically for virtual pins.

Related Information

Generating Design Partition Scripts for Project Management online help

Drive Clocks Directly

When partitions are exported from another Quartus II project, you should drive partition clock inputs directly with device clock input pins.

Connecting the clock signal directly avoids any timing analysis difficulties with gated clocks. Clock gating is never recommended for FPGA designs because of potential glitches and clock skew. Clock gating can be especially problematic with exported partitions because the partitions have no information about gating that takes place at the top-level design or in another partition. If a gated clock is required in a partition, perform the gating within that partition.

Direct connections to input clock pins also allows design partition scripts to send constraints from the top-level device pin to lower-level partitions.

Related Information

Invert Clocks in Destination Partitions on page 14-15

Recreate PLLs for Lower-Level Partitions if Required

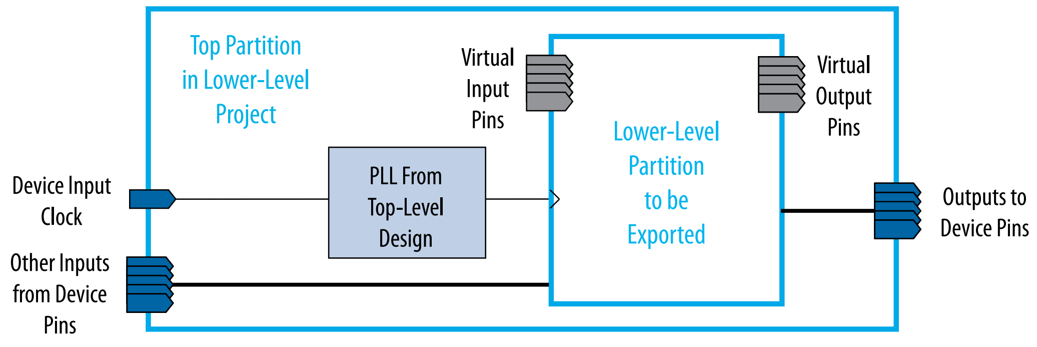
If you connect a PLL in your top-level design to partitions designed in separate Quartus II projects by third-party IP designers, the IP partitions do not have information about the multiplication, phase shift, or compensation delays for the PLL in the top-level design. To accommodate the PLL timing, you can make appropriate timing assignments in the projects created by IP designers to ensure that clocks are not left unconstrained or constrained with an incorrect frequency. Alternatively, you can duplicate the top-level PLL (or other derived clock logic) in the design file for the project created by the IP designer to ensure that you have the correct PLL parameters and clock delays for a complete and accurate timing analysis.

If the project lead creates a copy of the top-level project framework that includes all the settings and constraints needed for the design, this framework should include PLLs and other interface logic if this information is important to optimize partitions.

If you use a separate Quartus II project for an independent design block (such as when a designer or third-party IP provider does not have access to the entire design framework), include a copy of the top-level PLL in the lower-level partition as shown in figure.

In either case, the IP partition in the separate Quartus II project should contain just the partition logic that will be exported to the top-level design, while the full project includes more information about the top-level design. When the partition is complete, you can export just the partition without exporting the auxiliary PLL components to the top-level design. When you export a partition, the Quartus II software exports any hierarchy under the specified partition into the Quartus II Exported Partition File (**.qxp**), but does not include logic defined outside the partition (the PLL in this example).

Figure 14-19: Recreating a Top-Level PLL in a Lower-Level Partition


Top Partition in Lower-Level Project
Virtual Input Pins
Virtual Output Pins
Lower-Level Partition to be Exported
Device Input Clock
PLL From Top-Level Design
Outputs to Device Pins
Other Inputs from Device Pins

Checking Partition Quality

There are several tools you can use to create and analyze partitions in the Quartus II software. Take advantage of these tools to assess your partition quality, and use the information to improve your design or assignments as required to achieve the best results.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to ensure that your design follows Altera's recommendations for creating design partitions and implementing the incremental compilation design flow methodology. Each recommendation in the Incremental Compilation Advisor provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change.

Related Information

- **Incremental Compilation Advisor** on page 14-28
- **Incremental Compilation Advisor Command online help**
- **Example of Using the Incremental Compilation Advisor to Identify Non-Global Ports That Are Not Registered online help**
  For more information about the Incremental Compilation Advisor
- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation** on page 3-1

Design Partition Planner

The Design Partition Planner allows you to view design connectivity and hierarchy, and can assist you in creating effective design partitions that follow the guidelines in this manual. You can also use the Design Partition Planner to optimize design performance by isolating and resolving failing paths on a partition-by-partition basis.

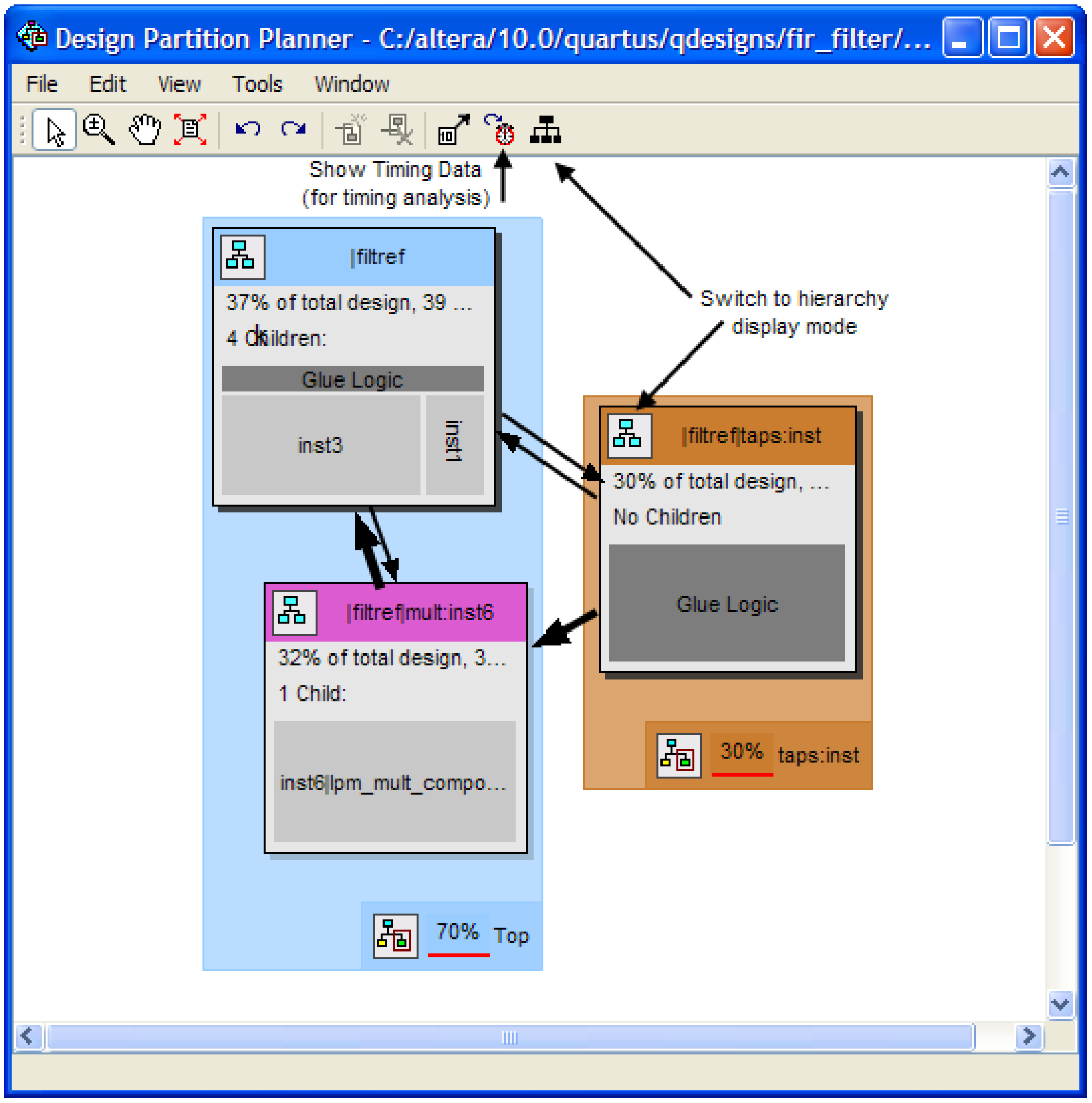
To view a design and create design partitions in the Design Partition Planner, you must first compile the design, or perform Analysis & Synthesis. In the Design Partition Planner, the design appears as a single top-level design block, with lower-level instances displayed as color-specific boxes.

In the Design Partition Planner, you can show connectivity between blocks and extract instances from the top-level design block. When you extract entities, connection bundles are drawn between entities, showing the number of connections existing between pairs of entities. When you have extracted a design block that you want to set as a design partition, right-click that design block, and then click **Create Design Partition**.

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

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

Figure 14-20: Design Partition Planner



You can switch between connectivity display mode and hierarchical display mode, to examine the view-only hierarchy display. You can also remove the connection lines between partitions and I/O banks by turning off **Display connections to I/O banks**, or use the settings on the **Connection Counting** tab in the **Bundle Configuration** dialog box to adjust how the connections are counted in the bundles.

To optimize design performance, confine failing paths within individual design partitions so that there are no failing paths passing between partitions. In the top-level entity, child entities that contain failing paths are marked by a small red dot in the upper right corner of the entity box.

To view the critical timing paths from a timing analyzer report, first perform a timing analysis on your design, and then in the Design Partition Planner, click **Show Timing Data** on the View menu.

Related Information

- Design Partition Planner online help
- Using the Design Partition Planner online help

Viewing Design Partition Planner and Floorplan Side-by-Side

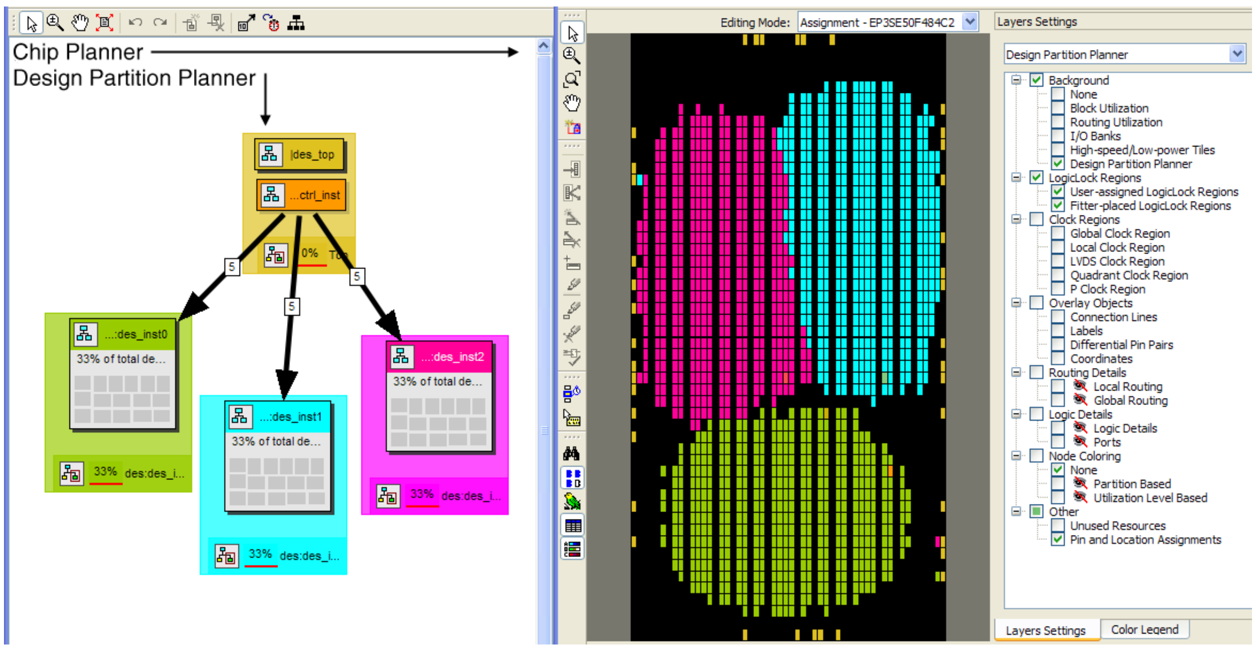
You can use the Design Partition Planner together with the Chip Planner to analyze natural placement groupings. This information can help you decide whether the design blocks should be grouped together in one partition, or whether they will make good partitions in the next compilation. It can also help determine whether the logic can easily be constrained by a LogicLock region. If logic naturally groups together when compiled without placement constraints, you can probably assign a reasonably sized LogicLock region to constrain the placement for subsequent compilations. You can experiment by extracting different design blocks in the Design Partition Planner and viewing the placement results of those design blocks from the previous compilation.

To view the Design Partition Planner and Chip Planner side-by-side, open the Design Partition Planner, and then open the Chip Planner and select the **Design Partition Planner** task. The **Design Partition Planner** task displays the physical locations of design entities with the same colors as in the Design Partition Planner.

In the Design Partition Planner, you can extract instances of interest from their parents by dragging and dropping, or with the **Extract from Parent** command. Evaluate the physical locations of instances in the Chip Planner and the connectivity between instances displayed in the Design Partition Planner. An entity is generally not suitable to be set as a separate design partition or constrained in a LogicLock region if the Chip Planner shows it physically dispersed over a noncontiguous area of the device after compilation. Use the Design Partition Planner to analyze the design connections. Child instances that are unsuitable to be set as separate design partitions or placed in LogicLock regions can be returned to their parent by dragging and dropping, or with the **Collapse to Parent** command.

The figure shows a design displayed in the Design Partition Planner and the Chip Planner with different colors for the top-level design and the three major design instances.

Figure 14-21: Design Partition Planner and Chip Planner



Partition Statistics Report

You can view statistics about design partitions in the Partition Merge Partition Statistics report and the **Statistics** tab of the **Design Partitions Properties** dialog box. These reports are useful when optimizing your design partitions, or when compiling the completed top-level design in a team-based compilation flow to ensure that partitions meet the guidelines discussed in this manual.

The Partition Merge Partition Statistics report in the Partition Merge section of the Compilation report lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells, as well as the number of input and output pins and how many are registered. This report also lists how many ports are unconnected, or driven by a constant V_{CC} or GND. You can use this information to assess whether you have followed the guidelines for partition boundaries.

You can also view statistics about the resource and port connections for a particular partition on the **Statistics** tab of the **Design Partition Properties** dialog box. The **Show All Partitions** button allows you to view all the partitions in the same report. The Partition Merge Partition Statistics report also shows statistics for the **Internal Congestion: Total Connections and Registered Connections**. This information represents how many signals are connected within the partition. It then lists the inter-partition connections for each partition, which helps you to see how partitions are connected to each other.

Related Information

Partition Merge Reports online help

Report Partition Timing in the TimeQuest Timing Analyzer

The Report Partitions diagnostic report and the `report_partitions` SDC command in the TimeQuest analyzer produce a **Partition Timing Overview** and **Partition Timing Details** table, which lists the partitions, the number of failing paths, and the worst case timing slack within each partition.

You can use these reports to analyze the location of the critical timing paths in the design in relation to partitions. If a certain partition contains many failing paths, or failing inter-partition paths, you might be able to change your partitioning scheme and improve timing performance.

Related Information

Quartus II TimeQuest Timing Analyzer documentation
Information about the TimeQuest `report_timing` command and reports

Check if Partition Assignments Impact the Quality of Results

You can ensure that you limit negative effect on the quality of results by following an iterative methodology during the partitioning process. In any incremental compilation flow where you can compile the source code for every partition during the partition planning phase, Altera recommends the following iterative flow:

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

   To run a full compilation, use the **Start Compilation** command.
2. Record the quality of results from the Compilation report (timing slack or f_{MAX}, area and any other relevant results).
3. Create design partitions following the guidelines described in this manual.
4. Recompile the design.
5. Record the quality of results from the Compilation report. If the quality of results is significantly worse than those obtained in the previous compilation, repeat step 3 through step 5 to change your partition assignments and use a different partitioning scheme.
6. Even if the quality of results is acceptable, you can repeat step 3 through step 5 by further dividing a large partition into several smaller partitions, which can improve compilation time in subsequent incremental compilations. You can repeat these steps until you achieve a good trade-off point (that is, all critical paths are localized within partitions, the quality of results is not negatively affected, and the size of each partition is reasonable).

You can also remove or disable partition assignments defined in the top-level design at any time during the design flow to compile the design as one flat compilation and get all possible design optimizations to assess the results. To disable the partitions without deleting the assignments, use the **Ignore partition assignments during compilation** option on the **Incremental Compilation** page of the **Settings** dialog box in the Quartus II software. This option disables all design partition assignments in your project and runs a full compilation, ignoring all partition boundaries and netlists. This option can be useful if you are using partitions to reduce compilation time as you develop various parts of the design, but can run a long compilation near the end of the design cycle to ensure the design meets its timing requirements.

Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery

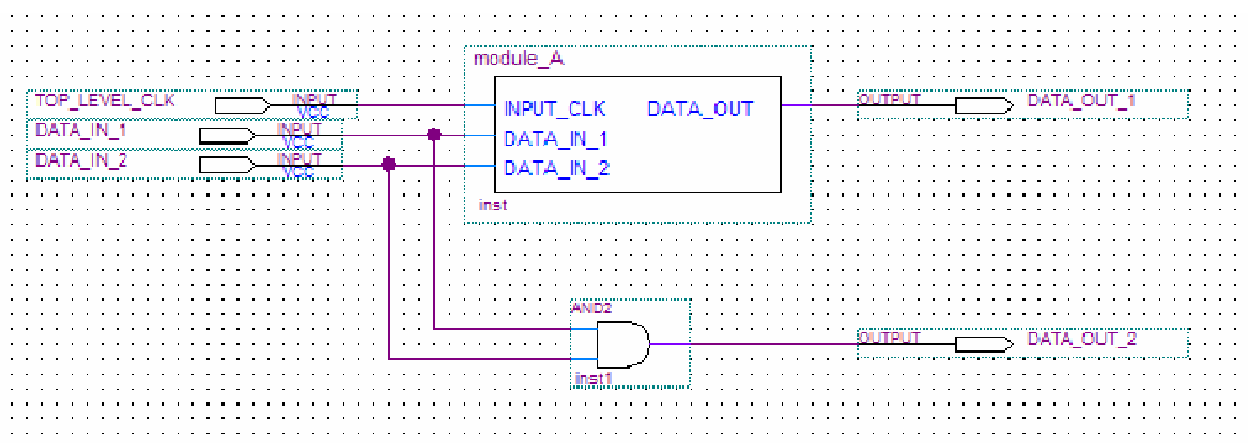
When exported partitions are compiled in a separate Quartus II project, such as when a third-party designer is delivering IP, the project lead must transfer the top-level project framework information and constraints to the partitions, so that each designer has a consistent view of the constraints that apply to the entire design. If the independent partition designers make any changes or add any constraints, they might have to transfer new constraints back to the project lead, so that these constraints are included in final timing sign-off of the entire design. Many assignments from the partition are carried with the partition into the top-level design; however, SDC format constraints for the TimeQuest analyzer are not copied into the top-level design automatically.

Passing additional timing constraints from a partition to the top-level design must be managed carefully. You can design within a single Quartus II project or a copy of the top-level design to simplify constraint management.

To ensure that there are no conflicts between the project lead's top-level constraints and those added by the third-party IP designer, use two **.sdc** files for each separate Quartus II project: an **.sdc** created by the project lead that includes project-wide constraints, and an **.sdc** created by the IP designer that includes partition-specific constraints.

The example design shown in the figure below is used to illustrate recommendations for managing the timing constraints in a third-party IP delivery flow. The top-level design instantiates a lower-level design block called `module_A` that is set as a design partition and developed by an IP designer in a separate Quartus II project.

Figure 14-22: Example Design to Illustrate SDC Constraints



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

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

Creating an .sdc File with Project-Wide Constraints

The **.sdc** with project-wide constraints for the separate Quartus II project should contain all constraints that are not completely localized to the partition. The **.sdc** should be maintained by the project lead. The project lead must ensure that these timing constraints are delivered to the individual partition owners and that they are syntactically correct for each of the separate Quartus II projects. This communication can be challenging when the design is in flux and hierarchies change. The project lead can use design partition scripts to automatically pass some of these constraints to the separate Quartus II projects.

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

The **.sdc** should include clock creation and clock constraints for any clock used by more than one partition. These constraints are particularly important when working with complex clocking structures, such as the following:

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

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

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

The project-wide **.sdc** can also contain any `set_input_delay` or `set_output_delay` constraints that are used for ports in separate Quartus II projects, because these represent delays external to a given partition. If the partition designer wants to set these constraints within the separate Quartus II projects, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.

Similarly, a constraint on a path that crosses a partition boundary should be in the project-wide **.sdc**, because it is not completely localized in a separate Quartus II project.

Related Information

Generating Design Partition Scripts for Project Management online help

Example Step 1—Project Lead Produces .sdc with Project-Wide Constraints for Lower-Level Partitions

The device input `top_level_clk` in **Figure 14-22** drives the `input_clk` port of `module_A`. To make sure the clock constraint is passed correctly to the partition, the project lead creates an **.sdc** with project-wide constraints for `module_A` that contains the following command:

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

The designer of `module_A` includes this **.sdc** as part of the separate Quartus II project.

Creating an .sdc with Partition-Specific Constraints

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

The partition-specific **.sdc** should be maintained by the partition designer; they must add any constraints required to properly compile and analyze their partition.

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

- Create a hierarchy variable for the partition (such as `module_A_hierarchy`) and set it to an empty string because the partition is the top-level instance in the separate Quartus II project. The project lead modifies this variable for the top-level hierarchy, reducing the effort of translating constraints on lower-level design hierarchies into constraints that apply in the top-level hierarchy. Use the following Tcl command first to check if the variable is already defined in the project, so that the top-level design does not use this empty hierarchy path: `if {![info exists module_A_hierarchy]}`.
- Use the hierarchy variable in the partition-specific **.sdc** as a prefix for assignments in the project. For example, instead of naming a particular instance of a register `reg:inst`, use `${module_A_hierarchy}reg:inst`. Also, use the hierarchy variable as a prefix to any wildcard characters (such as " * ").
- Pay attention to the location of the assignments to I/O ports of the partition. In most cases, these assignments should be specified in the **.sdc** with project-wide constraints, because the partition interface depends on the top-level design. If you want to set I/O constraints within the partition, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.
- Use caution with the `derive_clocks` and `derive_pll_clocks` commands. In most cases, the **.sdc** with project-wide constraints should call these commands. Because these commands impact the entire design, integrating them unexpectedly into the top-level design might cause problems.

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

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

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

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

Consolidating the .sdc in the Top-Level Design

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

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

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

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

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

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

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

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

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

Introduction to Design Floorplans

A floorplan represents the layout of the physical resources on the device. Creating a design floorplan, or floorplanning, describes the process of mapping the logical design hierarchy onto physical regions in the device.

In the Quartus II software, LogicLock regions can be used to constrain blocks of a design to a particular region of the device. LogicLock regions represent an area on the device with a user-defined or Fitter-defined size and location in the device layout.

Related Information

Analyzing and Optimizing the Design Floorplan with the Chip Planner documentation

The Difference between Logical Partitions and Physical Regions

Design partitions are logical entities based on the design hierarchy. LogicLock regions are physical placement assignments that constrain logic to a particular region on the device.

A common misconception is that logic from a design partition is always grouped together on the device when you use incremental compilation. Actually, logic from a partition can be placed anywhere in the device if it is not constrained to a LogicLock region, although the Fitter can pack related logic together to improve timing performance. A logical design partition does not refer to any physical area on the device and does not directly control where instances are placed on the device.

If you want to control the placement of logic from a design partition and isolate it to a particular part of the device, you can assign the logical design partition to a physical region in the device floorplan with a LogicLock region assignment. Altera recommends creating a design floorplan by assigning design partitions to LogicLock regions to improve the quality of results and avoid placement conflicts in some situations for incremental compilation.

Another misconception is that LogicLock assignments are used to preserve placement results for incremental compilation. Actually, LogicLock regions only constrain logic to a physical region on the device. Incremental compilation does not use LogicLock assignments or any location assignments to preserve the placement results; it simply reuses the results stored in the database netlist from a previous compilation.

Why Create a Floorplan?

Creating a design floorplan is usually required if you want to preserve placement for partitions that will be exported, to avoid resource conflicts between partitions in the top-level design. Floorplan location planning can be important for a design that uses incremental compilation, for the following reasons:

- To avoid resource conflicts between partitions, predominantly when integrating partitions exported from another Quartus II project.
- To ensure good quality of results when recompiling individual timing-critical partitions.

Location assignments for each partition ensure that there are no placement conflicts between partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating location, no device resources are specifically allocated for the logic associated with the region. If you do not clearly define resource allocation, logic placement can conflict when you integrate the partitions in the top-level design if you reuse the placement information from the exported netlist.

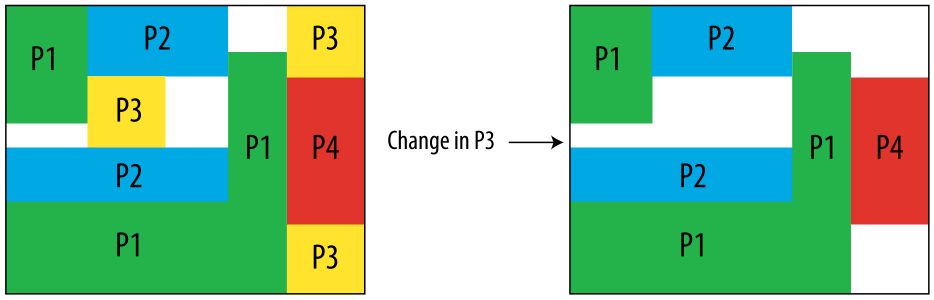
Creating a floorplan is also recommended for timing-critical partitions that have little timing margin to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions compiled in the same Quartus II project. The logic for partitions that are not timing-critical can be placed anywhere in the device on each recompilation if that is best for your design.

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

The figure illustrates the problems that may be associated with refitting designs that do not have floorplan location assignments. The left floorplan shows the initial placement of a four-partition design (P1-P4) without any floorplan location assignments. The right floorplan shows the device if a change occurs to P3. After removing the logic for the changed partition, the Fitter must re-place and reroute the new logic for P3 in the scattered white space. The placement of the post-fit netlists for other partitions forces the Fitter to implement P3 with the device resources that have not been used.

Figure 14-23: Representation of Device Floorplan without Location Assignments



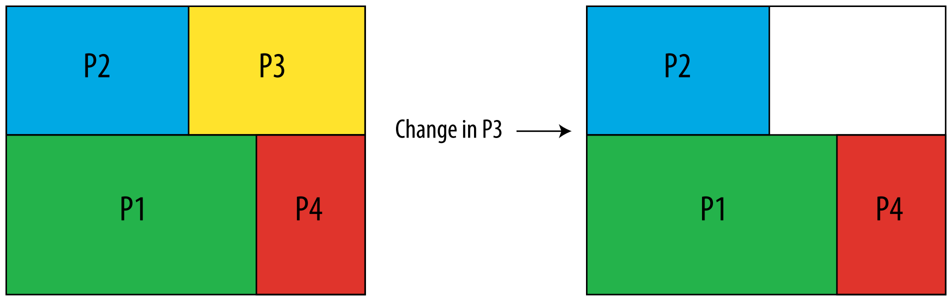
No floorplan assignments: Device has 4 partitions and the logic is placed throughout

Device after removing changed partition P3: New P3 must be placed in empty areas

The Fitter has a more difficult task because of more difficult physical constraints, and as a result, compilation time often increases. The Fitter might not be able to find any legal placement for the logic in partition P3, even if it could in the initial compilation. Additionally, if the Fitter can find a legal placement, the quality of results often decreases in these cases, sometimes dramatically, because the new partition is now scattered throughout the device.

The figure below shows the initial placement of a four-partition design with floorplan location assignments. Each partition is assigned to a LogicLock region. The second part of the figure shows the device after partition P3 is removed. This placement presents a much more reasonable task to the Fitter and yields better results.

Figure 14-24: Representation of Device Floorplan with Location Assignments



With floorplan location assignments: Device has 4 partitions placed in 4 LogicLock regions

Device after removing changed partition P3: Much easier to place new P3 partition in empty area

Altera recommends that you create a LogicLock floorplan assignment for timing-critical blocks with little timing margin that will be recompiled as you make changes to the design.

When to Create a Floorplan

It is important that you plan early to incorporate partitions into the design, and ensure that each partition follows partitioning guidelines. You can create floorplan assignments at different stages of the design flow, early or late in the flow. These guidelines help ensure better results as you begin creating floorplan location assignments.

Early Floorplan

An early floorplan is created before the design stage. You can plan an early floorplan at the top level of a design to allocate each partition a portion of the device resources. Doing so allows the designer for each block to create the logic for their design partition without conflicting with other logic. Each partition can be optimized in a separate Quartus II project if required, and the design can still be easily integrated in the top-level design. Even within one Quartus II project, each partition can be locked down with a post-fit netlist, and you can be sure there is space in the device floorplan for other partitions.

When you have compiled your complete design, or after you have integrated the first versions of partitions developed in separate Quartus II projects, you can use the design information and Quartus II features to tune and improve the floorplan .

Late Floorplan

A late floorplan is created or modified after the design is created, when the code is close to complete and the design structure is likely to remain stable. Creating a late floorplan is typically necessary only if you are starting to use incremental compilation late in the design flow, or need to reserve space for a logic block that becomes timing-critical but still has HDL changes to be integrated. When the design is complete, you can take advantage of the Quartus II analysis features to check the floorplan quality. To adjust the floorplan, you can perform iterative compilations as required and assess the results of different assignments.

Note: It may not be possible to create a good-quality late floorplan if you do not create partitions in the early stages of the design.

Design Floorplan Placement Guidelines

The following guidelines are key to creating a good design floorplan:

- Capture correct resources in each region.
- Use good region placement to maintain design performance compared to flat compilation.

A common misconception is that creating a floorplan enhances timing performance, as compared to a flat compilation with no location assignments. The Fitter does not usually require guidance to get optimal results for a full design.

Floorplan assignments can help maintain good performance when designs change incrementally. However, poor placement assignments in an incremental compilation can often adversely affect performance results, as compared to a flat compilation, because the assignments limit the options for the Fitter. Investing time to find good region placement is required to match the performance of a full flat compilation.

Flow for Creating a Floorplan

Use the following general procedure to create a floorplan:

1. Divide the design into partitions.
2. Assign the partitions to LogicLock regions.
3. Compile the design.
4. Analyze the results.
5. Modify the placement and size of regions, as required.

You might have to perform these steps several times to find the best combination of design partitions and LogicLock regions that meet the resource and timing goals of the design.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation** on page 3-1

Assigning Partitions to LogicLock Regions

Before compiling a design with new LogicLock assignments, ensure that the partition netlist type is set to **Post-Synthesis** or **Source File**, so that the Fitter does not reuse previous placement results.

In most cases, you should include logic from one partition in each LogicLock region. This organization helps to prevent resource conflicts when partitions are exported and can lead to better performance preservation when locking down parts of a design in a single project.

The Quartus II software is flexible and allows exceptions to this rule. For example, you can place more than one partition in the same LogicLock region if the partitions are tightly connected, but you do not want to merge the partitions into one larger partition. For best results, ensure that you recompile all partitions in the LogicLock region every time the logic in one partition changes. Additionally, if a partition contains multiple lower-level entities, you can place those entities in different areas of the device with multiple LogicLock regions, even if they are defined in the same partition.

You can use the **Reserved** LogicLock option to ensure that you avoid conflicts with other logic that is not locked into a LogicLock region. This option prevents other logic from being placed in the region, and is useful if you have empty partitions at any point during your design flow, so that you can reserve space in the floorplan. Do not make reserved regions too large to prevent unused area because no other logic can be placed in a region with the **Reserved** LogicLock option.

Related Information

LogicLock Region Properties Dialog Box online help

How to Size and Place Regions

In an early floorplan, assign physical locations based on design specifications. Use information about the connections between partitions, the partition size, and the type of device resources required.

In a late floorplan, when the design is complete, you can use locations or regions chosen by the Fitter as a guideline. If you have compiled the full design, you can view the location of the partition logic in the Chip Planner. You can use the natural grouping of each unconstrained partition as a starting point for a LogicLock region constraint. View the placement for each partition that requires a floorplan constraint, and create a new LogicLock region by drawing a box around the area on the floorplan, and then assigning the partition to the region to constrain the partition placement.

Instead of creating regions based on the previous compilation results, you can start with the Fitter results for a default auto size and floating origin location for each new region when the design logic is complete. After compilation, lock the size and origin location.

Alternatively, if the design logic is complete with auto-sized or floating location regions, you can specify the size based on the synthesis results and use the locations chosen by the Fitter with the **Set to Estimated Size** command. Like the previous option, start with floating origin location. After compilation, lock the origin location. You can also enable the **Fast Synthesis Effort** setting to reduce synthesis time.

After a compilation, save the Fitter size and origin location of the Fitter with the **Set Size and Origin to Previous Fitter Results** command.

Note: It is important that you use the Fitter-chosen locations only as a starting point to give the regions a good fixed size and location. Ensure that all LogicLock regions in the design have a fixed size and have their origin locked to a specific location on the device. On average, regions with fixed size and location yield better timing performance than auto-sized regions.

Related Information

- Checking Partition Quality on page 14-28
- Creating and Manipulating LogicLock Regions online help

Modifying Region Size and Origin

After saving the Fitter results from an initial compilation for a late floorplan, modify the regions using your knowledge of the design to set a specific size and location. If you have a good understanding of how the design fits together, you can often improve upon the regions placed in the initial compilation. In an early floorplan, when the design has not yet been created, you can use the guidelines in this section to set the size and origin, even though there is no initial Fitter placement.

The easiest way to move and resize regions is to drag the region location and borders in the Chip Planner. Make sure that you select the **User-Defined** region in the floorplan (as opposed to the **Fitter-Placed** region from the last compilation) so that you can change the region.

Generally, you can keep the Fitter-determined relative placement of the regions, but make adjustments if required to meet timing performance. Performing a full compilation ensures that the Fitter can optimize for a full placement and routing.

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

If resource utilization is low in the overall device, enlarge the regions. Doing so usually improves the final results because it gives the Fitter more freedom to place additional or modified logic added to the partition during subsequent incremental compilations. It also allows room for optimizations such as pipelining and physical synthesis logic duplication.

Try to have each region evenly full, with the same ”fullness” that the complete design would have without LogicLock regions; Altera recommends approximately 75% full.

Allow more area for regions that are densely populated, because overly congested regions can lead to poor results. Allow more empty space for timing-critical partitions to improve results. However, do not make regions too large for their logic. Regions that are too large can result in wasted resources and also lead to suboptimal results.

Ideally, almost the entire device should be covered by LogicLock regions if all partitions are assigned to regions.

Regions should not overlap in the device floorplan. If two partitions are allocated on an overlapping portion of the chip, each may independently claim common resources in this region. This leads to resource conflicts when integrating results into a top-level design. In a single project, overlapping regions give more difficult constraints to the Fitter and can lead to reduced quality of results.

You can create hierarchical LogicLock regions to ensure that the logic in a child partition is physically placed inside the LogicLock region for its parent partition. This can be useful when the parent partition does not contain registers at the boundary with the lower-level child partition and has a lot of signal connectivity. To create a hierarchical relationship between regions in the LogicLock Regions window, drag and drop the child region to the parent region.

I/O Connections

Consider I/O timing when placing regions. Using I/O registers can minimize I/O timing problems, and using boundary registers on partitions can minimize problems connecting regions or partitions. However, I/O timing might still be a concern. It is most important for flows where each partition is compiled independently, because the Fitter can optimize the placement for paths between partitions if the partitions are compiled at the same time.

Place regions close to the appropriate I/O, if necessary. For example, DDR memory interfaces have very strict placement rules to meet timing requirements. Incorporate any specific placement requirements into your floorplan as required. You should create LogicLock regions for internal logic only, and provide pin location assignments for external device I/O pins (instead of including the I/O cells in a LogicLock region to control placement).

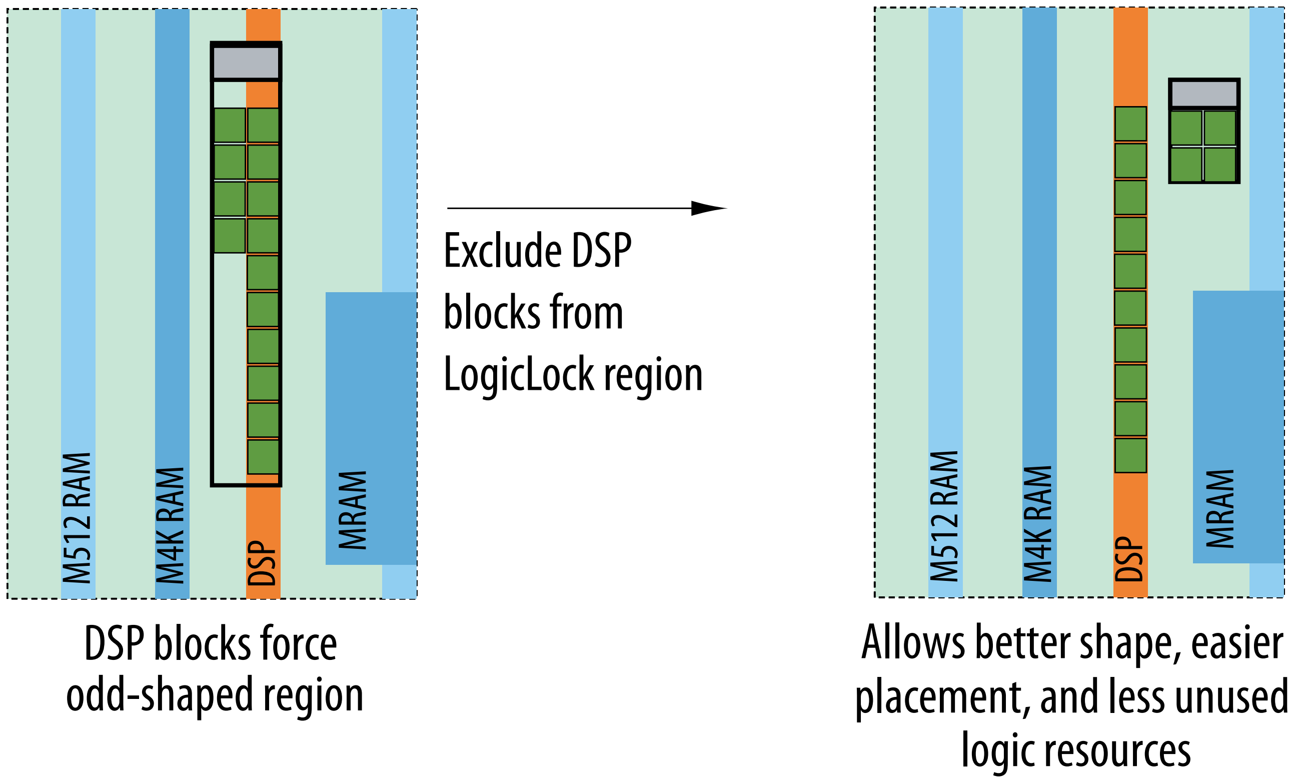
LogicLock Resource Exclusions

You can exclude certain resource types from a LogicLock region to manage the ratio of logic to dedicated DSP and RAM resources in the region.

If your design contains memory or Digital Signal Processing (DSP) elements, you may want to exclude these elements from the LogicLock region. LogicLock resource exceptions prevent certain types of elements from being assigned to a region. Therefore, those elements are not required to be placed inside the region boundaries. The option does not prevent them from being placed inside the region boundaries unless the **Reserved** property of the region is turned on.

Resource exceptions are useful in cases where it is difficult to place rectangular regions for design blocks that contain memory and DSP elements, due to their placement in columns throughout the device floorplan. Exclude RAMs, DSPs, or logic cells to give the Fitter more flexibility with region sizing and placement. Excluding RAM or DSP elements can help to resolve no-fit errors that are caused by regions spanning too many resources, especially for designs that are memory-intensive, DSP-intensive, or both. The figure shows an example of a design with an odd-shaped region to accommodate DSP blocks for a region that does not contain very much logic. The right side of the figure shows the result after excluding DSP blocks from the region. The region can be placed more easily without wasting logic resources.

Figure 14-25: LogicLock Resource Exclusion Example


M512 RAM
M4K RAM
DSP
MRAM
Exclude DSP blocks from LogicLock region
DSP blocks force odd-shaped region
M512 RAM
M4K RAM
DSP
MRAM
Allows better shape, easier placement, and less unused logic resources

To view any resource exceptions, right-click in the LogicLock Regions window, and then click **LogicLock Regions Properties**. In the **LogicLock Regions Properties** dialog box, select the design element (module or entity) in the **Members** box, and then click **Edit**. In the **Edit Node** dialog box, to set up a resource exception, click the **Edit** button next to the **Excluded element types** box, and then turn on the design element types to be excluded from the region. You can choose to exclude combinational logic or registers from logic cells, or any of the sizes of TriMatrix memory blocks, or DSP blocks.

If the excluded logic is in its own lower-level design entity (even if it is within the same design partition), you can assign the entity to a separate LogicLock region to constrain its placement in the device.

You can also use this feature with the LogicLock **Reserved** property to reserve specific resources for logic that will be added to the design.

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

To assign a code block to a LogicLock region, with exclusions, use the following command:

```
set_logiclock_contents -region <LogicLock region name> \
-to <block> -exceptions \"<keyword>:<keyword>"
```

- *<LogicLock region name>*—The name of the LogicLock region to which the code block is assigned.
- *<block>*—A code block in a Quartus II project hierarchy, which can also be a design partition.
- *<keyword>*—The list of exceptions made during assignment. For example, if DSP was in the keyword list, the named block of code would be assigned to the LogicLock region, except for any DSP block within the code block. You can include the following exceptions in the `set_logiclock_contents` command:

Keyword variables:

- *REGISTER*—Any registers in the logic cells.
- *COMBINATIONAL*—Any combinational elements in the logic cells.
- *SMALL_MEM*—Small TriMatrix memory blocks (M512 or MLAB).
- *MEDIUMEM_MEM*—Medium TriMatrix memory blocks (M4K or M9K).
- *LARGE_MEM*—Large TriMatrix memory blocks (M-RAM or M144K).
- *DSP*—Any DSP blocks.
- *VIRTUAL_PIN*—Any virtual pins.

Note: Resource filtering uses the optional Tcl argument `-exclude_resources` in the `set_logiclock_contents` function. If left unspecified, no resource filter is created. In the **.qsf**, resource filtering uses an extra LogicLock membership assignment called `LL_MEMBER_RESOURCE_EXCLUDE`. For example, the following line in the **.qsf** is used to specify a resource filter for the `alu:alu_unit` entity assigned to the ALU region.

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

Creating Non-Rectangular Regions

To constrain placement to non-rectangular or non-contiguous areas of the device, you can connect multiple rectangular regions together using the **Merge** command.

For devices that do not support the **Merge** command (MAX™ II devices), you can limit entity placement to a sub-area of a LogicLock region to create non-rectangular constraints. In these devices, construct a LogicLock hierarchy by creating child regions inside of parent regions, and then use the **Reserved** option to control which logic can be placed inside these child regions. Setting the **Reserved** option for the region prevents the Fitter from placing nodes that are not assigned to the region inside the boundary of the region.

Related Information

Creating and Manipulating LogicLock Regions online help

Checking Floorplan Quality

The Quartus II software has several tools to help you create a floorplan. You can use these tools to assess your floorplan quality and use the information to improve your design or assignments as required to achieve the best results.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows the recommendations for creating floorplan location assignments that are presented in this manual.

LogicLock Region Resource Estimates

You can view resource estimates for a LogicLock region to determine the region's resource coverage, and use this estimate before compilation to check region size. Using this estimate helps to ensure adequate resources when you are sizing or moving regions.

Related Information

LogicLock Region Properties Dialog Box online help

LogicLock Region Properties Statistics Report

LogicLock region statistics are similar to design partition properties, but also include resource usage details after compilation.

The statistics report the number of resources used and the total resources covered by the region, and also lists the number of I/O connections and how many I/Os are registered (good), as well as the number of internal connections and the number of inter-region connections (bad).

Locate the Quartus II TimeQuest Timing Analyzer Path in the Chip Planner

In the TimeQuest analyzer user interface, you can locate a specific path in the Chip Planner to view its placement and perform a report timing operation (for example, report timing for all paths with less than 0 ns slack).

Related Information

Locate Dialog Box online help
Information about how to locate paths between the TimeQuest analyzer and the Chip Planner

Inter-Region Connection Bundles

The Chip Planner can display bundles of connections between LogicLock regions, with filtering options that allow you to choose the relevant data for display. These bundles can help you to visualize how many connections there are between each LogicLock region to improve floorplan assignments or to change partition assignments, if required.

Related Information

Inter-region Bundles Dialog Box online help
Information about how to display bundles of connections between LogicLock regions

Routing Utilization

The Chip Planner includes a feature to display a color map of routing congestion. This display helps identify areas of the chip that are too tightly packed.

In the Chip Planner, red LAB blocks indicate higher routing congestion. You can position the mouse pointer over a LAB to display a tooltip that reports the logic and routing utilization information.

Related Information

Chip Planner online help
Information about how to how to view a color map of routing congestion in the Chip Planner

Ensure Floorplan Assignments Do Not Significantly Impact Quality of Results

The end results of design partitioning and floorplan creation differ from design to design. However, it is important to evaluate your results to ensure that your scheme is successful. Compare your before and after results, and consider using another scheme if any of the following guidelines are not met:

- You should see only minor degradation in f_{MAX} after the design is partitioned and floorplan location assignments are created. There is some performance cost associated with setting up a design for incremental compilation; approximately 3% is typical.
- The area increase should be no more than 5% after the design is partitioned and floorplan location assignments are created.
- The time spent in the routing stage should not significantly increase.

The amount of compilation time spent in the routing stage is reported in the Messages window with an Info message that indicates the elapsed time for Fitter routing operations. If you notice a dramatic increase in routing time, the floorplan location assignments may be creating substantial routing congestion. In this case, decrease the number of LogicLock regions, which typically reduces the compilation time in subsequent incremental compilations and may also improve design performance.

Recommended Design Flows and Application Examples

Listed below are application examples with design flows for partitioning and creating a design floorplan during common timing closure and team-based design scenarios. Each flow describes the situation in which it should be used, and provides a step-by-step description of the commands required to implement the flow.

Create a Floorplan for Major Design Blocks

Use this incremental compilation flow for designs when you want to assign a floorplan location for each major block in your design. A full floorplan ensures that partitions do not interact as they are changed and recompiled— each partition has its own area of the device floorplan.

To create a floorplan for major design blocks, follow this general methodology:

1. In the Design Partitions window, ensure that all partitions have their netlist type set to **Source File** or **Post-Synthesis**. If the netlist type is set to **Post-Fit**, floorplan location assignments are not used when recompiling the design.
2. Create a LogicLock region for each partition (including the top-level entity, which is set as a partition by default).
3. Run a full compilation of your design to view the initial Fitter-chosen placement of the LogicLock regions as a guideline.
4. In the Chip Planner, view the placement results of each partition and LogicLock region on the device.
5. If required, modify the size and location of the LogicLock regions in the Chip Planner. For example, enlarge the regions to fill up the device and allow for future logic changes.You can also, if needed, create a new LogicLock region by drawing a box around an area on the floorplan.
6. Run the Compiler with the **Start Compilation** command to determine the timing performance of your design with the modified or new LogicLock regions.
7. Repeat steps 5 and 6 until you are satisfied with the quality of results for your design floorplan. Once you are satisfied with your results, run a full compilation of your design.

Create a Floorplan Assignment for One Design Block with Difficult Timing

Use this flow when you have one timing-critical design block that requires more optimization than the rest of your design. You can take advantage of incremental compilation to reduce your compilation time without creating a full design floorplan.

In this scenario, you do not want to create floorplan assignments for the entire design. Instead, you can create a region to constrain the location of your critical design block, and allow the rest of the logic to be placed anywhere on the device. To create a region for critical design block, follow these steps:

1. Divide up your design into partitions. Ensure that you isolate the timing-critical logic in a separate partition.
2. Define a LogicLock region for the timing-critical partition. Ensure that you capture the correct amount of device resources in the region. Turn on the **Reserved** property to prevent any other logic from being placed in the region.

    - If the design block is not complete, reserve space in the design floorplan based on your knowledge of the design specifications, connectivity between design blocks, and estimates of the size of the partition based on any initial implementation numbers.
    - If the critical design block has initial source code ready, compile the design to place the LogicLock region. Save the Fitter-determined size and origin, and then enlarge the region to provide more flexibility and allow for future design changes.

    As the rest of the design is completed, and the device fills up, the timing-critical region reserves an area of the floorplan. When you make changes to the design block, the logic will be re-placed in the same part of the device, which helps ensure good quality of results.

Related Information

Design Partition Guidelines on page 14-9

Create a Floorplan as the Project Lead in a Team-Based Flow

Use this approach when you have several designs that will be implemented in separate Quartus II projects by different designers, or third-party IP designers who want to optimize their designs independently and pass the results to the project lead.

As the project lead in this scenario, follow these steps to prepare the top-level design for a successful team-based design methodology with early floorplan planning:

1. Create a new Quartus II project that will ultimately contain the full implementation of the entire design.
2. Create a “skeleton” or framework of the design that defines the hierarchy for the subdesigns that will be implemented by separate designers. Consider the partitioning guidelines in this manual when determining the design hierarchy.
3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.
4. Make design partition assignments for each major subdesign. Set the netlist type for each partition that will be implemented in a separate Quartus II project and later exported and integrated with the top-level design set to **Empty**.
5. Create LogicLock regions for each partition to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any

initial implementation numbers and knowledge of the design specifications. Use the guidelines described in this chapter to choose a size and location for each LogicLock region.

6. Provide the constraints from the top-level design to partition designers using one of the following procedures:
    a. Create a copy of the top-level Quartus II project framework by checking out the appropriate files from a source control system, using the **Copy Project** command, or creating a project archive. Provide each partition designer with the copy of the project.
    b. Provide the constraints with documentation or scripts.

Related Information

Generating Design Partition Scripts for Project Management online help

Document Revision History

Table 14-1: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	Removed support for early timing estimate feature.
2014.12.15	14.1.0	• Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings. • Updated description of Virtual Pin assignment to clarify that assigned pins are no longer free as input pins.
June 2014	14.0.0	• Dita conversion. • Removed obsolete devices content for Arria GX, Cyclone, Cyclone II, Cyclone III, Stratix, Stratix GX, Stratix II, Stratix II GX, • Replace Megafunction content with IP Catalog and Parameter Editor content.
November 2013	13.1.0	Removed HardCopy device information.
November 2012	12.1.0	Added Turning On Supported Cross-Boundary Optimizations.
June 2012	12.0.0	Removed survey link.
November 2011	11.0.1	Template update.
May 2011	11.0.0	Updated links.

Date	Versio n	Changes
December 2010	10.1.0	• Changed to new document template. • Moved "Creating Floorplan Location Assignments With Tcl Commands—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks)" from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the *Quartus II Handbook.* • Consolidated Design Partition Planner and Incremental Compilation Advisor information between the Quartus II Incremental Compilation for Hierarchical and Team-Based Design and Best Practices for Incremental Compilation Partitions and Floorplan Assignments handbook chapters.
July 2010	10.0.0	• Removed the explanation of the "bottom-up design flow" where designers work completely independently, and replaced with Altera's recommendations for team-based environments where partitions are developed in the same top-level project framework, plus an explanation of the bottom-up process for including independent partitions from third-party IP designers. • Expanded the **Merge** command explanation to explain how it now accommodates cross-partition boundary optimizations. • Restructured Altera recommendations for when to use a floorplan.
October 2009	9.1.0	• Redefined the bottom-up design flow as team-based and reorganized previous design flow examples to include steps on how to pass top-level design information to lower-level projects. • Added "Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery" from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the *Quartus II Handbook.* • Reorganized the "Recommended Design Flows and Application Examples" section. • Removed HardCopy APEX and HardCopy Stratix Devices section.
March 2009	9.0.0	• Added I/O register packing examples from *Incremental Compilation for Hierarchical and Team-Based Designs* chapter • Moved "Incremental Compilation Advisor" section • Added "Viewing Design Partition Planner and Floorplan Side-by-Side" section • Updated Figure 15-22 • Chapter 8 was previously Chapter 7 in software release 8.1.
November 2008	8.1.0	• Changed to 8-1/2 x 11 page size. No change to content.

Date	Versio n	Changes
May 2007	8.0.0	• Initial release.

Related Information

Quartus II Handbook Archive

For previous versions of the Quartus II Handbook

Mitigating Single Event Upsets 15

2014.06.30

QII5V1 Subscribe Send Feedback

The Quartus II software offers several features to enable the detection and correct ion of single event upsets (SEUs), or soft errors, as well as to characterize the effects of SEU on your designs.

Understanding SEU

SEU can affect any semiconductor device.

SEUs are rare, unintended changes in the state of internal memory elements, caused by cosmic radiation effects. The change in state results in a soft error, so the affected device can be reset to its original value and there is no permanent damage to the device itself. Because of the unintended memory state, the device may operate erroneously until this upset is fixed.

The Soft Error Rate (SER) is expressed as Failure-in-Time (FIT) units, defined as one soft error occurrence every billion hours of operation. Often SEU mitigation is not required because of the low chance of occurrence. However, for highly complex systems, such as with multiple high-density components, error rate may be a significant system design factor. If your system includes multiple FPGAS and requires very high reliability and availability, you should consider the implications of soft errors, and use the available techniques for detecting and recovering from these types of errors. If your system is requiring high reliability and availability, consider the implications of soft errors, and use the techniques in this document to detect and recover from these types of errors.

FPGAs use memory both in user logic (bulk memory and registers) and in Configuration Random Access Memory (CRAM). CRAM configures the FPGA; this is the memory loaded with the contents of a **.sof** file by the Quartus II Programmer. The CRAM configures all logic and routing in the device. If an SEU strikes a CRAM bit, the effect can be harmless if the CRAM bit is not in use. However, the affect can be severe if it affects critical logic internal signal routing (such as a lookup table bit).

Related Information

Introduction to Single Event Upsets

Mitigating SEU Effects in Embedded User RAM

You can mitigate SEU effects for internal memories by using Error Correcting Codes (ECC) for internal SRAM blocks. This method is effective enough so that the FIT rate of ECC-protected memories is almost zero.

ECC use a Cyclic Redundancy Check (CRC) code for a given data word. These CRC bits provide redundancy on the data word which can detect the location of single-bit and double-bit flips in the data word. Since the location of the bit flips is known, the CRC can locate and correct the error. The CRC code can also account for bit flips in the CRC code itself. Altera FPGAs support both Single Error Correction Double Error Detection (SECDED) and Double Error Correction Triple Error Detection (DECTED), depending on the device family.

Some Altera device memory blocks offer built-in CRC circuitry hardened in silicon. This is available in the M20K memory in Stratix V, and in the M144K block in Stratix IV, and Arria II devices. (Other device families can implement CRC functions using Altera IP cores). The CRC circuitry generates an EDCRC code at the data storage input of the RAM, and checks the CRC code at the output of the RAM. If an SEU affects any stored bits in the internal memory, the CRC automatically corrects the error when it is read from the memory. The ECC-enabled memory can report the occurrence of a single-bit flip, or adjacent double-bit and adjacent triple-bit flips, and will correct single- and double-bit flips. Adjacent triple-bit corruptions are detected and reported using a status bit, but not corrected.

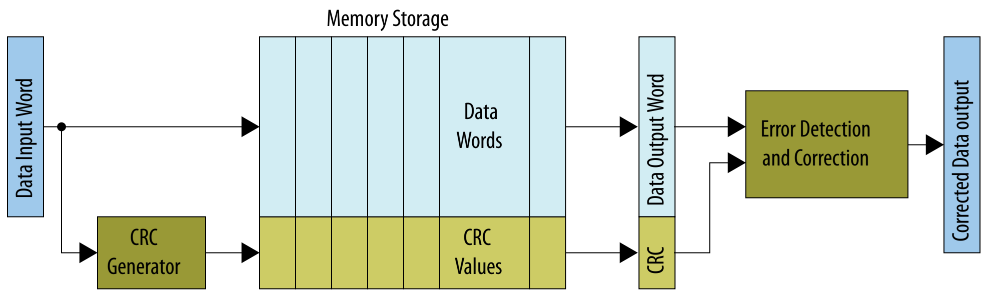
Configuring the ECCRAM

You must configure the ECCRAM as a 2-port RAM (with independent read and write addresses). Use of these features does not reduce the amount of available logic.

While the CRC checking function results in some additional output delay, the hard ECC has a much higher f_{MAX} compared with an equivalent soft ECC implemented in general logic. Additionally, the hard IP can be pipelined in the M20K block by configuring the ECCRAM to use an output register at the corrected data output port. This increases performance while adding latency.

For devices without dedicated circuitry, you can implement the ECC by instantiating the ECC generation and checking functions as the IP core ALTECC.

Figure 15-1: Memory Storage

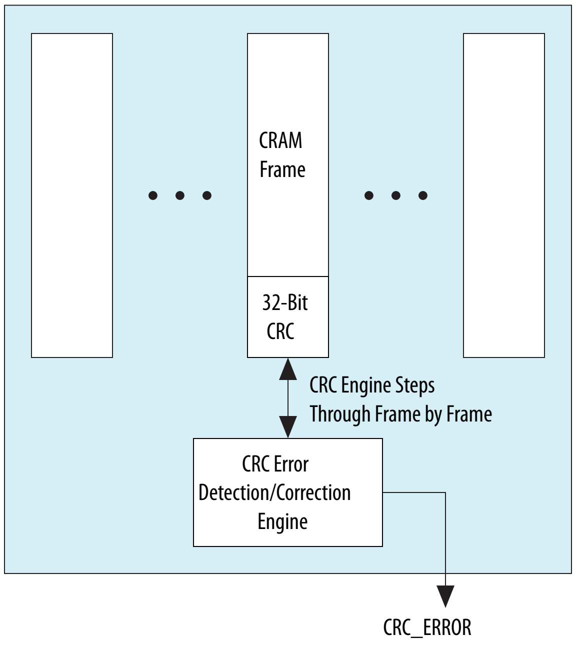


Mitigating SEU Effects in Configuration RAM

Use EDCRC to detect and correct soft errors in CRAM. These EDCRC blocks are similar to those that protect internal user memory.

CRAM is organized into frames. The size of the frame and the number of frames is device specific. CRAM frames are continually checked for errors by loading each frame into a data register. The EDCRC block checks the frame for errors. Soft errors found trigger the assertion of a CRC_ERROR pin on the device. Monitor this pin in your system. Take appropriate actions when this pin is asserted, indicating a soft error was detected in the configuration RAM.

Figure 15-2: CRAM Frame


CRAM Frame
32-Bit CRC
CRC Engine Steps Through Frame by Frame
CRC Error Detection/Correction Engine
CRC_ERROR

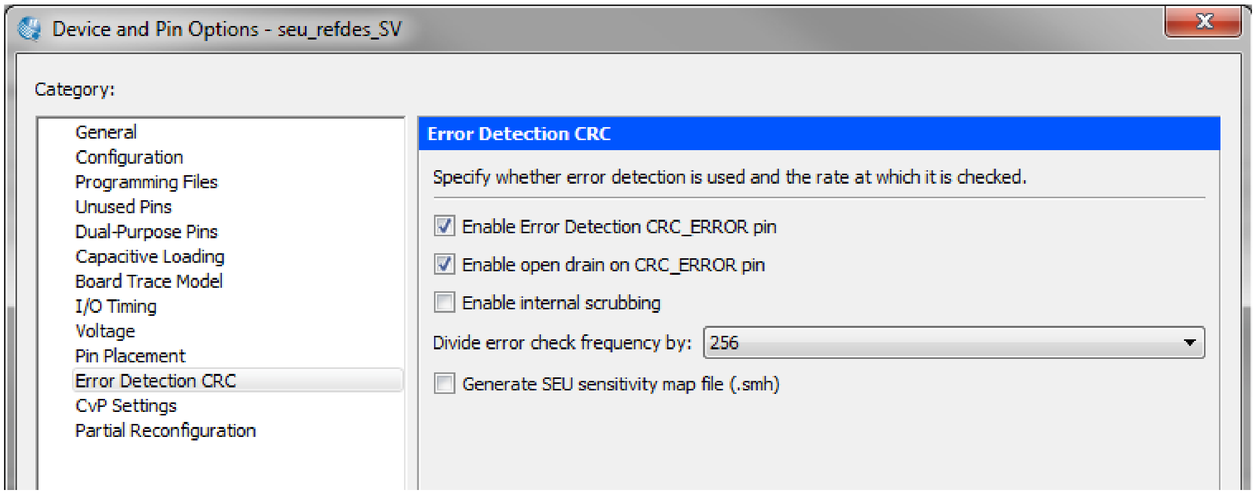
Related Information

Single Event Upsets

Scanning CRAM Frames

To enable the Quartus II software to scan CRAM frames, turn on **Enable Error Detection CRC_ERROR** pin in the **Device and Pin Options** dialog box (**Assignments** > **Device** > **Device and Pin Options**).

Figure 15-3: Enable Error Detection CRC_ERROR Pin



Error Detection CRC

Specify whether error detection is used and the rate at which it is checked.

[✓] Enable Error Detection CRC_ERROR pin
[✓] Enable open drain on CRC_ERROR pin
[ ] Enable internal scrubbing
Divide error check frequency by: 256
[ ] Generate SEU sensitivity map file (.smh)

To enable the CRC_ERROR pin as an open drain output, turn on **Enable open drain on CRC_ERROR pin**.

To guarantee the availability of a clock, the EDCRC function operates on an independent clock generated internally on the FPGA itself. To enable EDCRC operation on a divided version of the clock select a value from the **Divide error check frequency by** value.

Internal Scrubbing

Arria V, Cyclone V (including SoC devices), Stratix V, and later device families support automatic CRAM error correction, without resorting to the original CRAM contents from an external copy of the original SRAM Object File.

Automatic correction is possible because EDCRC calculates and stores redundancy fields along with the configuration bits. This automatic correction is known as scrubbing.

To enable internal scrubbing, turn on **Enable internal scrubbing** option in the **Device and Pin Options** dialog box.

If the Quartus II software finds a CRC error in a CRAM frame, the frame is reconstructed from the error correcting code calculated for that frame, and then the corrected frame is re-written into the CRAM.

Note: If you enable internal scrubbing, you must still plan a recovery sequence. Although scrubbing can restore the CRAM array to intended configuration, latency occurs between the soft error detection and correction. Because of the large number of configuration bits to be scanned, this latency may be up to 100 milliseconds for large devices. Therefore, the FPGA may operate with errors during that period.

Related Information

Error Detection CRC Page

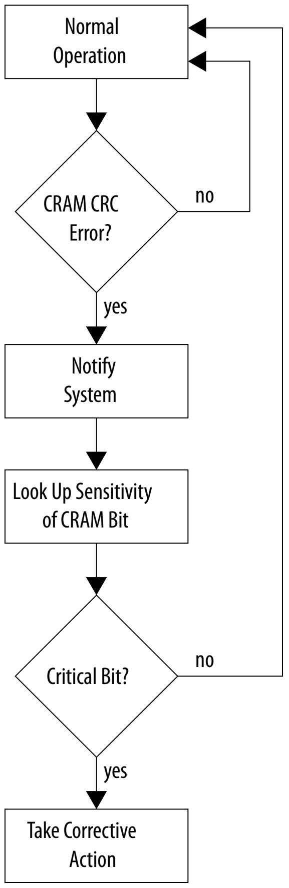
Understanding SEU Sensitivity

Reconfigurating a running FPGA typically has a significant impact on the system using the FPGA. When planning for SEU recovery, account for the time required to bring the FPGA to a state consistent with the current state of the system. For example, if an internal state machine is in an illegal state, it mayrequire reset. Also, the surrounding logic may need to account for this unexpected operation.

Often an SEU impacts CRAM bits not used by the implemented design. Many configuration bits are not used because they control logic and routing wires that are not used in a design. Depending on the implementation, 40% of all CRAM bits can be used even in the most heavily utilized devices. This means that only 40% of SEU events require intervention, and you can ignore 60% of SEU events.

You may determine that portions of the implemented design are not critical to the FPGA's function. Examples may include test circuitry implemented but not important to the operation of the device, or other non-critical functions that may be logged but do not need to be reprogrammed or reset.

Figure 15-4: Sensitivity Processing Flow



The ratio of SEU strikes versus functional interrupts is the Single Event Functional Interrupt (SEFI) ratio. Minimizing this ratio improves SEU mitigation.

Related Information

Understanding Single Event Functional Interrupts in FPGA Designs

Designating the Sensitivity of your Design Hierarchy

The design hierarchy sensitivity processing depends on the contents of the Sensitivity Map Header File (**.smf**). This file determines the correct (least disruptive) recovery sequence for any CRAM bit flip. The **.smf** designates the sensitivity of each portion of the FPGA's logic design.

To generate the **.smf**, you must designate the sensitivity of the design from a functional logic view, using the hierarchy tagging procedure.

Hierarchy Tagging

Hierarchy tagging is the process of classifying the sensitivity of the portions of your design.

The Quartus II software performs hierarchy tagging by creating a design partition, and then assigning the parameter `ASD Region` to that partition. The parameter can assume a value from 0 to 255, so there are 256 different classifications of system responses to the portions of your design. This sensitivity information is encoded into the **.smf** the running system uses to look-up the sensitivity of an SEU upset, and to perform the appropriate action to that CRAM location.

Related Information

Altera Advanced SEU Detection IP Core User Guide

Altera Advanced SEU Detection IP Core

You must instantiate the Altera Advanced SEU Detection IP core to enable SEU detection and correction features.

When the EDCRC function detects an SEU, the Altera Advanced SEU Detection IP core determines the designer-designated sensitivity of that CRAM bit by looking up the sensitivity in the **.smf**.

When an EDCRC block detects an SEU, a sensitivity processor looks up the sensitivity of the affected CRAM bit in the **.smf**.

The user determines which version of the IP core to instantiate: on-chip or external. If the Altera Advanced SEU Detection IP core is configured for on-chip sensitivity processing, the IP core performs the lookup with the user-supplied memory interface. If the Altera Advanced SEU Detection IP core is configured for off-chip sensitivity processing, it notifies external logic (typically via a system CPU interrupt request), and provides cached event message register values to the off-chip sensitivity processor. The SMH information is stored in the external sensitivity processor's memory system.

Related Information

Altera Advanced SEU Detection IP Core User Guide

On-Chip Sensitivity Processor

You can use the Advanced SEU Detection IP core to implement an on-chip sensitivity processor. The IP core interacts with user-supplied external memory access logic to read the Sensitivity Map Header file, stored on external memory.

Once it determines the sensitivity of the affected CRAM bit, the IP core can assert a Critical Error signal so the system provides an appropriate response. If the SEU is not critical, the Critical Error signal may be left un-asserted.

On-chip sensitivity processing is autonomous: the FPGA device determines whether it is affected by an SEU, without the need for external logic. However, this requires part of the FPGA's logic resources for the external memory interface.

Related Information

Altera Advanced SEU Detection IP Core User Guide

External Sensitivity Processor

You can configure the Advanced SEU Detection IP core for use with an external sensitivity processor. I n this case an external CPU, such as the ARM processor in Altera's SoC devices, receives an interrupt request when the FPGA detects an SEU. The CPU then reads the Error Message Register, and performs the sensitivity lookup by referring to the Sensitivity Map Header file (**.smf**) stored in the CPU's memory space.

External sensitivity processing does not require on-board memory dedicated to the SMH storage function,. Also, this technique relieves the FPGA of external memory interface requirements, along with the memory storage requirements for the sensitivity map itself. If a CPU is already present in the system, external sensitivity processing may be the more hardware-efficient way to implement sensitivity lookup.

Related Information

Altera Advanced SEU Detection IP Core User Guide

Triple-Module Redundancy

If your system must suffer no downtime due to SEUs, consider Triple Module Redundancy as an SEU mitigation strategy.

Triple-Module-Redundancy (TMR) is an established technique for improving hardware fault tolerance. In TMR, three identical instances of hardware are supplied, along with voting hardware at the output of the hardware. If an SEU affects one of the instances, the voting logic notes the majority in a vote of the separate instances of the module to mask out any malfunctioning module.

The advantage of TMR is that there is no downtime in the case of a single SEU; if a module is found to be in faulty operation, that module can be scrubbed of its error by reprogramming it. The error detection and correction time is many orders of magnitude less than the MTBF due to SEU events. Therefore, you can repair a soft interrupt before another SEU affects another instance in the TMR triple.

The disadvantage of TMR is its extreme cost in hardware resources: it requires three times as much hardware, in addition to voting logic. This hardware cost can be minimized by judiciously implementing TMR only for the most critical part of the design.

There are several automated ways to generate TMR designs by automatically replicating designated functions and synthesizing the required voting logic. Synthesis vendors offering automated TMR synthesis include Synopsys and Mentor Graphics.

Recovering from a Single-Event Upset

After correcting a bit flip in CRAM, the device is in its original configuration with respect to logic and routing. However, the internal state of the FPGA may be illegal.

The state of the device may be invalid because it may have been operating while SEUs corrupted its configuration. The errors from faulty operation may have propagated elsewhere within the FPGA or to the system outside the FPGA.

Forcing the FPGA into a known state is system dependent. Determining the possible outcomes from SEU, and designing a recovery response to SEU should be part of the FPGA and system design process.

Evaluating Your System's Response to Functional Upsets

Because SEUs can randomly strike any memory element, system testing is especially important to ensure a comprehensive recovery response.

The Quartus II software includes the Fault Injection Debugger to aid in SEU recovery response. This feature is available for the Arria V, Cyclone V, and Stratix V device families.

The feature is available from the Quartus II GUI or at the command line. You must instantiate the Altera Fault Injection IP core into your FPGA design to use this feature. The IP core flips a CRAM bit by dynamically reconfiguring the frame containing that CRAM bit, flipping it to its opposite state.

The Fault Injection Debugger allows you to operate the FPGA in your system and inject random CRAM bit flips to test the ability of the FPGA and the system to detect and recover fully from an SEU. You should be able to observe your FPGA and your system recover from these simulated SEU strikes. You can then refine your FPGA and system recovery sequence by observing these strikes.

If you have recorded an SEU in the device's Error Message Register, the Fault Injection Debugger also allows you to specify a targeted fault to be injected (rather than inject the fault in a random location). This feature is available only from the command line.

Related Information

Debugging Single Event Upsets Using the Fault Injection Debugger

Document Revision History

Table 15-1: Document Revision History

Date	Version	Changes
June 2014	2014.06.30	• Updated formatting. • Added "Mitigating SEU Effects in Embedded User RAM" section. • Added "Altera Advanced SEU Detection IP Core" section.
November 2012	2012.11.01	Preliminary release.

Quartus II Integrated Synthesis 16

2015.05.04

QII5V1 Subscribe Send Feedback

As programmable logic designs become more complex and require increased performance, advanced synthesis becomes an important part of a design flow. The Altera® Quartus® II software includes advanced Integrated Synthesis that fully supports VHDL, Verilog HDL, and Altera-specific design entry languages, and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use solution.

Related Information

Recommended HDL Coding Styles on page 12-1
For examples of Verilog HDL and VHDL code synthesized for specific logic functions

Designing With Low-Level Primitives User Guide
For more information about coding with primitives that describe specific low-level functions in Altera devices

Design Flow

The Quartus II Analysis & Synthesis stage of the compilation flow runs Integrated Synthesis, which fully supports Verilog HDL, VHDL, and Altera-specific languages, and major features of the SystemVerilog language.

In the synthesis stage of the compilation flow, the Quartus II software performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic in device resources such as logic elements (LEs) or adaptive logic modules (ALMs), and other dedicated logic blocks. The synthesis stage generates a single project database that integrates all your design files in a project (including any netlists from third-party synthesis tools).

You can use Analysis & Synthesis to perform the following compilation processes:

Table 16-1: Compilation Process

Compilation Process	Description
Analyze Current File	Parses your current design source file to check for syntax errors. This command does not report many semantic errors that require further design synthesis. To perform this analysis, on the Processing menu, click **Analyze Current File**.

Compilation Process	Description
Analysis & Elaboration	Checks your design for syntax and semantic errors and performs elaboration to identify your design hierarchy. To perform Analysis & Elaboration, on the Processing menu, point to **Start,** and then click **Start Analysis & Elaboration**.
Hierarchy Elaboration	Parses HDL designs and generates a skeleton of hierarchies. Hierarchy Elaboration is similar to the Analysis & Elaboration flow, but without any elaborated logic, thus making it much faster to generate.
Analysis & Synthesis	Performs complete Analysis & Synthesis on a design, including technology mapping. To perform Analysis & Synthesis, on the Processing menu, point to **Start,** and then click **Start Analysis & Synthesis**.

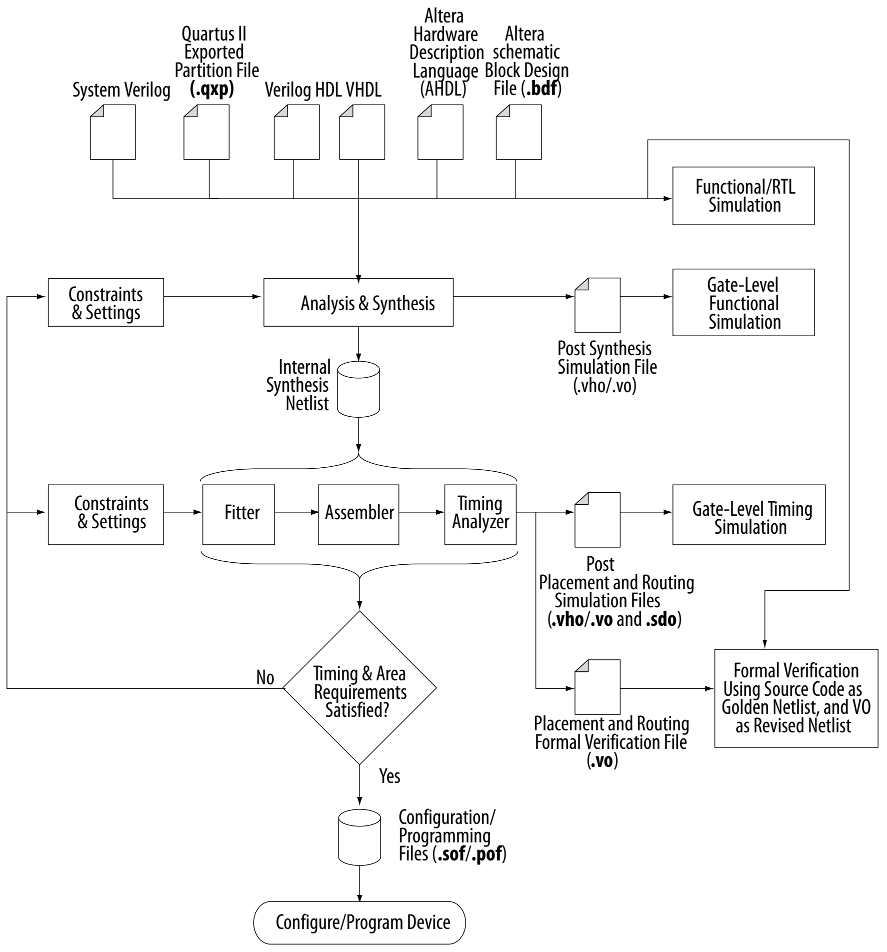
Related Information

Language Support on page 16-4

Start Hierarchy Elaboration Command Processing Menu
For more information about the Hierarchy Elaboration flow

Quartus II Integrated Synthesis Design and Compilation Flow

Figure 16-1: Basic Design Flow Using Quartus II Integrated Synthesis


System Verilog
Quartus II Exported Partition File (.qxp)
Verilog HDL
VHDL
Altera Hardware Description Language (AHDL)
Altera schematic Block Design File (.bdf)
Functional/RTL Simulation
Constraints & Settings
Analysis & Synthesis
Post Synthesis Simulation File (.vho/.vo)
Gate-Level Functional Simulation
Internal Synthesis Netlist
Constraints & Settings
Fitter
Assembler
Timing Analyzer
Gate-Level Timing Simulation
Post Placement and Routing Simulation Files (.vho/.vo and .sdo)
No
Timing & Area Requirements Satisfied?
Yes
Placement and Routing Formal Verification File (.vo)
Formal Verification Using Source Code as Golden Netlist, and VO as Revised Netlist
Configuration/ Programming Files (.sof/.pof)
Configure/Program Device

The Quartus II Integrated Synthesis design and compilation flow consists of the following steps:

1. Create a project in the Quartus II software and specify the general project information, including the top-level design entity name.
2. Create design files in the Quartus II software or with a text editor.
3. On the Project menu, click **Add/Remove Files in Project** and add all design files to your Quartus II project using the **Files** page of the **Settings** dialog box.
4. Specify Compiler settings that control the compilation and optimization of your design during synthesis and fitting.
5. Add timing constraints to specify the timing requirements.
6. Compile your design. To synthesize your design, on the Processing menu, point to **Start**, and then click **Start Analysis & Synthesis**. To run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis, click **Start Compilation** on the Processing menu.
7. After obtaining synthesis and placement and routing results that meet your requirements, program or configure your Altera device.

Integrated Synthesis generates netlists that enable you to perform functional simulation or gate-level timing simulation, timing analysis, and formal verification.

Related Information

- **Quartus II Synthesis Options** on page 16-21
  For more information about synthesis settings
- **Incremental Compilation** on page 16-20
  For more information about partitioning your design to reduce compilation time
- **Quartus II Exported Partition File as Source** on page 16-21
  For more information about using **.qxp** as a design source file
- **Introduction to the Quartus II Software**
  For an overall summary of features in the Quartus II software
- **Managing Files in a Project**
  For more information about Quartus II projects
- **About Compilation Flows**
  For more information about Quartus II the compilation flow

Language Support

Quartus II Integrated Synthesis supports HDL. You can specify the Verilog HDL or VHDL language version in your design.

To ensure that the Quartus II software reads all associated project files, add each file to your Quartus II project by clicking **Add/Remove Files in Project** on the Project menu. You can add design files to your project. You can mix all supported languages and netlists generated by third-party synthesis tools in a single Quartus II project.

Related Information

- **Design Libraries** on page 16-12
  Describes how to compile and reference design units in custom libraries
- **Using Parameters/Generics** on page 16-15
  Describes how to use parameters or generics and pass them between languages
- **Insert Template Dialog Box**
  For more information about using the available templates in the Quartus II Text Editor for various Verilog and VHDL features

Verilog HDL Support

The Quartus II Compiler's Analysis & Synthesis module supports the following Verilog HDL standards:

- Verilog-1995 (IEEE Standard 1364-1995)
- Verilog-2001 (IEEE Standard 1364-2001)
- SystemVerilog-2005 (IEEE Standard 1800-2005) (the Compiler does not support all constructs)

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified. The Quartus II Compiler uses the Verilog-2001 standard by default for files that have the extension **.v**, and the SystemVerilog standard for files that have the extension **.sv**.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file.

The Quartus II software support for Verilog HDL is case sensitive in accordance with the Verilog HDL standard. The Quartus II software supports the compiler directive `` `define``, in accordance with the Verilog HDL standard.

The Quartus II software supports the `include` compiler directive to include files with absolute paths (with either "/" or "\" as the separator), or relative paths. When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the Quartus II software cannot find the file, the software then searches relative to all user libraries and then relative to the directory location of the current file.

Related Information

- **About Verilog HDL**
  For more information about Verilog HDL
- **Quartus II Verilog HDL Support**
  For more information about Quartus II Verilog HDL support
- **Specifying Verilog Input Settings**
  For more information about specifying a default Verilog HDL version for all files
- **verilog_input_version Synthesis Directive**
  For more information about controlling the Verilog HDL version that compiles your design in a design file with the VERILOG_INPUT_VERSION synthesis directive
- **Verilog HDL Synthesis Attributes and Directives**
  For more information about Verilog HDL synthesis attributes and directives
- **Adding an HDL File to a Project and Setting the HDL Version** on page 16-76
- **Synthesis Directives** on page 16-25
  For more information about specifying synthesis directives

Verilog HDL Configuration

Verilog HDL configuration is a set of rules that specify the source code for particular instances.

Verilog HDL configuration allows you to perform the following tasks:

- Specify a library search order for resolving cell instances (as does a library mapping file)
- Specify overrides to the logical library search order for specified instances
- Specify overrides to the logical library search order for all instances of specified cells

For more information about these tasks, refer to **Table 16-2**.

Configuration Syntax

A Verilog HDL configuration contains the following statements:

```
config config_identifier;
design [library_identifier.]cell_identifier;
config_rule_statement;
endconfig
```

Where:

- `config`—the keyword that begins the configuration.
- `config_identifier`—the name you enter for the configuration.
- `design`—the keyword that starts a design statement for specifying the top of the design.
- `[library_identifier.]cell_identifier`—specifies the top-level module (or top-level modules) in the design and the logical library for this module (modules).
- `config_rule_statement`—one or more of the following clauses: default, instance, or cell. For more information, refer to Table 16-2.
- `endconfig`—the keyword that ends a configuration.

Table 16-2: Type of Clauses for the config_rule_statement Keyword

Clause Type	Description
`default`	Specifies the logical libraries to search to resolve a default cell instance. A default cell instance is an instance in the design that is not specified in a subsequent instance or cell clause in the configuration. You specify these libraries with the `liblist` keyword. The following is an example of a default clause: `default liblist lib1 lib2;` Also specifies resolving default instances in the logical libraries (`lib1` and `lib2`). Because libraries are inherited, some simulators (for example, VCS) also search the default (or current) library as well after the searching the logical libraries (`lib1` and `lib2`).

`instance`	Specifies a specific instance. The specified instance clause depends on the use of the following keywords:
	• `liblist`—specifies the logical libraries to search to resolve the instance.
	• `use`—specifies that the instance is an instance of the specified cell in the specified logical library.
	
	The following are examples of `instance` clauses:
	
	`instance top.dev1 liblist lib1 lib2;`
	
	This instance clause specifies to resolve `instance top.dev1` with the cells assigned to logical libraries `lib1` and `lib2`;
	
	`instance top.dev1.gm1 use lib2.gizmult;`
	
	This instance clause specifies that `top.dev1.gm1` is an instance of the cell named `gizmult` in logical library `lib2`.

`cell`	A `cell` clause is similar to an `instance` clause, except that the `cell` clause specifies all instances of a cell definition instead of specifying a particular instance. What it specifies depends on the use of the `liblist` or `use` keywords: • `liblist`—specifies the logical libraries to search to resolve all instances of the cell. • `use`—the specified cell's definition is in the specified library.

Hierarchical Configurations

A design can have more than one configuration. For example, you can define a configuration that specifies the source code you use in particular instances in a sub hierarchy, then define a configuration for a higher level of the design.

Suppose, for example, a sub hierarchy of a design is an eight-bit adder and the RTL Verilog code describes the adder in a logical library named `rtllib` and the gate-level code describes the adder in a logical library named `gatelib`.

If you want to use the gate-level code for the 0 (zero) bit of the adder and the RTL level code for the other seven bits, the configuration might appear as shown in the following example:

```
config cfg1;
design aLib.eight_adder;
default liblist rtllib;
instance adder.fulladd0 liblist gatelib;
endconfig
```

If you are instantiating this eight-bit adder eight times to create a 64-bit adder, use configuration `cfg1` for the first instance of the eight-bit adder, but not in any other instance. A configuration that would perform this function is shown in the following example:

```
config cfg2;
design bLib.64_adder;
default liblist bLib;
instance top.64add0 use work.cfg1:config;
endconfig
```

Note: The name of the unbound module may be different than the name of the cell that is bounded to the instance.

Suffix :config

To distinguish between a module by the same name, use the optional extension `:config` to refer to configuration names. For example, you can always refer to a `cfg2` configuration as `cfg2:config` (even if the `cfg2` module does not exist).

SystemVerilog Support

The Quartus II software supports the SystemVerilog constructs.

Note: Designs written to support the Verilog-2001 standard might not compile with the SystemVerilog setting because the SystemVerilog standard has several new reserved keywords.

Related Information

- **Quartus II Support for SystemVerilog**
  For more information about the supported SystemVerilog constructs
- **Quartus II Support for Verilog 2001**
  For more information about the supported Verilog-2001 features

Initial Constructs and Memory System Tasks

The Quartus II software infers power-up conditions from the Verilog HDL `initial` constructs. The Quartus II software also creates power-up settings for variables, including RAM blocks. If the Quartus II software encounters nonsynthesizable constructs in an `initial` block, it generates an error.

To avoid such errors, enclose nonsynthesizable constructs (such as those intended only for simulation) in `translate_off` and `translate_on` synthesis directives

Synthesis of initial constructs enables the power-up state of the synthesized design to match the power-up state of the original HDL code in simulation.

Note: Initial blocks do not infer power-up conditions in some third-party EDA synthesis tools. If you convert between synthesis tools, you must set your power-up conditions correctly.

Quartus II Integrated Synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories.

This example shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 16-1: Verilog HDL Code: Initializing RAM with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
$readmemb("ram.txt", ram);
end
```

When creating a text file to use for memory initialization, specify the address using the format @<location> on a new line, and then specify the memory word such as `110101` or `abcde` on the next line.

The following example shows a portion of a Memory Initialization File (.mif) for the RAM in Example 16-1.

Example 16-2: Text File Format: Initializing RAM with the readmemb Command

```
@0
00000000
@1
00000001
@2
00000010
…
@e
00001110
@f
00001111
```

Related Information

- Translate Off and On / Synthesis Off and On on page 16-58
- Power-Up Level on page 16-36

Verilog HDL Macros

The Quartus II software fully supports Verilog HDL macros, which you can define with the `'define` compiler directive in your source code. You can also define macros in the Quartus II software or on the command line.

Setting a Verilog HDL Macro Default Value in the Quartus II Software

To specify a macro in the Quartus II software, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Verilog HDL Input**
2. Under **Verilog HDL macro**, type the macro name in the **Name** box and the value in the **Setting** box.
3. Click **Add**.

Setting a Verilog HDL Macro Default Value on the Command Line

To set a default value for a Verilog HDL macro on the command line, use the `--verilog_macro` option:

```
quartus_map <Design name> --verilog_macro= "<Macro name>=<Macro setting>"
```

The command in this example has the same effect as specifying `` `define a 2`` in the Verilog HDL source code:

```
quartus_map my_design --verilog_macro="a=2"
```

To specify multiple macros, you can repeat the option more than once.

```
quartus_map my_design --verilog_macro="a=2" --verilog_macro="b=3"
```

VHDL Support

The Quartus II Compiler's Analysis & Synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)
- VHDL 2008 (IEEE Standard 1076-2008)

The Quartus II Compiler uses the VHDL 1993 standard by default for files that have the extension **.vhdl** or **.vhd**.

Note: The VHDL code samples follow the VHDL 1993 standard.

To specify a default VHDL version for all files, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Verilog HDL Input**
2. On the **VHDL Input** page, under **VHDL version**, select the appropriate version, and then click **OK**.

   To override the default VHDL version for each VHDL design file, follow these steps:
3. On the Project menu, click **Add/Remove Files in Project**.
4. On the **Files** page, select the appropriate file in the list, and then click **Properties**.
5. In the HDL version list, select **VHDL_2008**, **VHDL_1993**, or **VHDL_1987,** and then click **OK**.

You can also specify the VHDL version that compiles your design for each design file with the `VHDL_INPUT_VERSION` synthesis directive. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Table 16-3: Controlling the VHDL Input Version with a Synthesis Directive

HDL	Code
VHDL	`--synthesis VHDL_INPUT_VERSION <language version>`

HDL	Code
VHDL-2008	`/* synthesis VHDL_INPUT_VERSION <language version> */`

The variable <language version> requires one of the following values:

- `VHDL_1987`
- `VHDL_1993`
- `VHDL_2008`

When the Quartus II software reads a `VHDL_INPUT_VERSION` synthesis directive, it changes the current language version as specified until after the file or until it reaches the next `VHDL_INPUT_VERSION` directive.

Note: You cannot change the language version in a VHDL design unit.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file.

The Quartus II software reads default values for registered signals defined in the VHDL code and converts the default values into power-up level settings. This enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.

Related Information

- **Synthesis Directives** on page 16-25
  For more information about specifying synthesis directives
- **Adding an HDL File to a Project and Setting the HDL Version** on page 16-76
  For more information about using the `-HDL_VERSION` command to specify the HDL version for each design file
- **Power-Up Level** on page 16-36
  For more information about power-up level

VHDL-2008 Support

The Quartus II software contains support for VHDL 2008 with constructs defined in the IEEE Standard 1076-2008 version of the *IEEE Standard VHDL Language Reference Manual*.

Related Information

Quartus II Support for VHDL 2008
For more information about the Quartus II software support for VHDL-2008

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and several vendor-specific VHDL libraries.

The IEEE library includes the standard VHDL packages `std_logic_1164`, `numeric_std`, `numeric_bit`, and `math_real`. The STD library is part of the VHDL language standard and includes the packages standard (included in every project by default) and textio. For compatibility with older designs, the Quartus II software also supports the following vendor-specific packages and libraries:

- Synopsys packages such as std_logic_arith and std_logic_unsigned in the IEEE library
- Mentor Graphics® packages such as std_logic_arith in the ARITHMETIC library
- Altera primitive packages altera_primitives_components (for primitives such as GLOBAL and DFFE) and maxplus2 (for legacy support of MAX+PLUS® II primitives) in the ALTERA library
- Altera IP core packages altera_mf_components and stratixgx_mf_components in the ALTERA_MF library (for Altera-specific IP cores including LCELL), and lpm_components in the LPM library for library of parameterized modules (LPM) functions.

Note: Altera recommends that you import component declarations for Altera primitives such as GLOBAL and DFFE from the altera_primitives_components package and not the altera_mf_components package.

Related Information

- **Design Libraries** on page 16-12
  For information about organizing your own design units into custom libraries

VHDL wait Constructs

The Quartus II software supports one VHDL wait until statement per process block. However, the Quartus II software does not support other VHDL wait constructs, such as wait for and wait on statements, or processes with multiple wait statements.

The following shows the wait until construct example for VHDL:

```
architecture dff_arch of ls_dff is
begin
output: process begin
wait until (CLK'event and CLK='1');
Q <= D;
Qbar <= not D;
end process output;
end dff_arch;
```

AHDL Support

The Quartus II Compiler's Analysis & Synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (**.tdf**). You can import AHDL Include Files (**.inc**) into a **.tdf** with an AHDL include statement. Altera provides **.inc** files for all IP cores shipped with the Quartus II software.

Note: The AHDL language does not support the synthesis directives or attributes.

Related Information

About AHDL
For more information about AHDL

Schematic Design Entry Support

The Quartus II Compiler's Analysis & Synthesis module fully supports **.bdf** for schematic design entry.

Note: Schematic entry methods do not support the synthesis directives or attributes.

Related Information

About Schematic Design Entry
For information about creating and editing schematic designs

State Machine Editor

The Quartus II software supports graphical state machine entry. To create a new finite state machine (FSM) design, on the File menu, click **New**. In the **New** dialog box, expand the **Design Files** list, and then select **State Machine File**.

Related Information

About the State Machine Editor
For more information about the State Machine Editor

Design Libraries

By default, the Quartus II software compiles all design files into the work library. If you do not specify a design library, if a file refers to a library that does not exist, or if the referenced library does not contain a referenced design unit, the Quartus II software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup, but you have the option of creating separate custom design libraries.

To compile your design files into specific libraries (for example, when you have two or more functionally different design entities that share the same name), you can specify a destination library for each design file in various ways, as described in the following:

- **Specifying a Destination Library Name in the Settings Dialog Box** on page 16-13
- **Specifying a Destination Library Name in the Quartus II Settings File or with Tcl** on page 16-13

When the Quartus II Compiler analyzes the file, it stores the analyzed design units in the destination library of the file.

Note: A design can contain two or more entities with the same name if the Quartus II software compiles the entities into separate libraries.

When compiling a design instance, the Quartus II software initially searches for the entity in the library associated with the instance (which is the work library if you do not specify any library). If the Quartus II software could not locate the entity definition, the software searches for a unique entity definition in all design libraries. If the Quartus II software finds more than one entity with the same name, the software generates an error. If your design uses multiple entities with the same name, you must compile the entities into separate libraries.

In VHDL, you can associate an instance with an entity in several ways, as described in **Mapping a VHDL Instance to an Entity in a Specific Library** on page 16-14.

In Verilog HDL, BDF schematic entry, AHDL, VQM and EDIF netlists, you can use different libraries for each of the entities that have the same name, and compile the instantiation into the same library as the appropriate entity.

Related Information

Mapping a VHDL Instance to an Entity in a Specific Library on page 16-14

Specifying a Destination Library Name in the Settings Dialog Box

To specify a library name for one of your design files, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Files**.
3. Select the file in the **File Name** list.
4. Click **Properties**.
5. In the **File Properties** dialog box, select the type of design file from the **Type** list.
6. Type the library name in the **Library** field.
7. Click **OK**.

Specifying a Destination Library Name in the Quartus II Settings File or with Tcl

You can specify the library name with the `-library` option to the <language type>`_FILE` assignment in the Quartus II Settings File (**.qsf**) or with Tcl commands.

For example, the following assignments specify that the Quartus II software analyzes the **my_file.vhd** and stores its contents (design units) in the VHDL library **my_lib**, and then analyzes the Verilog HDL file **my_header_file.h** and stores its contents in a library called **another_lib**.

```
set_global_assignment -name VHDL_FILE my_file.vhd -library my_lib
set_global_assignment -name VERILOG_FILE my_header_file.h -library another_lib
```

Related Information

- Scripting Support on page 16-75
  For more information about Tcl scripting

Specifying a Destination Library Name in a VHDL File

You can use the `library` synthesis directive to specify a library name in your VHDL source file. This directive takes the name of the destination library as a single string argument. Specify the `library` directive in a VHDL comment before the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), with one of the supported keywords for synthesis directives, that is, `altera`, `synthesis`, `pragma`, `synopsys`, or `exemplar`.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file in the **Settings** dialog box, any existing **.qsf** setting, any setting made through the Tcl interface, or any prior `library` directive in the current file. The directive remains effective until the end of the file or the next `library` synthesis directive.

The following example uses the `library` synthesis directive to create a library called **my_lib** containing the `my_entity` design unit:

```
-- synthesis library my_lib
library ieee;
use ieee.std_logic_1164.all;
entity my_entity(...)
end entity my_entity;
```

Note: You can specify a single destination library for all your design units in a given source file by specifying the library name in the **Settings** dialog box, editing the **.qsf**, or using the Tcl interface. To organize your design units in a single file into different libraries rather than just a single library, you can use the `library` directive to change the destination VHDL library in a source file.

The Quartus II software generates an error if you use the library directive in a design unit.

Related Information

- Synthesis Directives on page 16-25
  For more information about specifying synthesis directives

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides several ways to map or bind an instance to an entity in a specific library.

Direct Entity Instantiation

In the direct entity instantiation method, the instantiation refers to an entity in a specific library.

The following shows the direct entity instantiation method for VHDL:

```
entity entity1 is
port(...);
end entity entity1;
architecture arch of entity1 is
begin
inst: entity lib1.foo
port map(...);
end architecture arch;
```

Component Instantiation—Explicit Binding Instantiation

You can bind a component to an entity in several mechanisms. In an explicit binding indication, you bind a component instance to a specific entity.

The following shows the binding instantiation method for VHDL:

```
entity entity1 is
port(...);
end entity entity1;
package components is
component entity1 is
port map (...);
end component entity1;
end package components;
entity top_entity is
port(...);
end entity top_entity;
use lib1.components.all;
architecture arch of top_entity is
-- Explicitly bind instance I1 to entity1 from lib1
for I1: entity1 use entity lib1.entity1
port map(...);
end for;
begin
I1: entity1 port map(...);
end architecture arch;
```

Component Instantiation—Default Binding

If you do not provide an explicit binding indication, the Quartus II software binds a component instance to the nearest visible entity with the same name. If no such entity is visible in the current scope, the Quartus II software binds the instance to the entity in the library in which you declare the component. For example, if you declare the component in a package in the `MY_LIB` library, an instance of the component binds to the entity in the `MY_LIB` library.

The code examples in the following examples show this instantiation method:

Example 16-3: VHDL Code: Default Binding to the Entity in the Same Library as the Component Declaration

```
use mylib.pkg.foo; -- import component declaration from package “pkg” in


                                -- library “mylib”
architecture rtl of top
...
begin
-- This instance will be bound to entity “foo” in library “mylib”
inst: foo
port map(...);
end architecture rtl;
```

Example 16-4: VHDL Code: Default Binding to the Directly Visible Entity

```
use mylib.foo; -- make entity “foo” in library “mylib” directly visible
architecture rtl of top
component foo is
generic (...)
port (...);
end component;
begin
-- This instance will be bound to entity “foo” in library “mylib”
inst: foo
port map(...);
end architecture rtl;
```

Using Parameters/Generics

The Quartus II software supports parameters (known as generics in VHDL) and you can pass these parameters between design languages.

Click **Assignments** > **Settings** > **Compiler Settings** > **Default Parameters** to enter default parameter values for your design. In AHDL, the Quartus II software inherits parameters, so any default parameters apply to all AHDL instances in your design. You can also specify parameters for instantiated modules in a **.bdf**. To specify parameters in a **.bdf** instance, double-click the parameter value box for the instance symbol, or right-click the symbol and click **Properties**, and then click the **Parameters** tab.

You can specify parameters for instantiated modules in your design source files with the provided syntax for your chosen language. Some designs instantiate entities in a different language; for example, they might instantiate a VHDL entity from a Verilog HDL design file. You can pass parameters or generics between VHDL, Verilog HDL, AHDL, and BDF schematic entry, and from EDIF or VQM to any of these languages. You do not require an additional procedure to pass parameters from one language to another. However, sometimes you must specify the type of parameter you are passing. In those cases, you must follow certain guidelines to ensure that the Quartus II software correctly interprets the parameter value.

Related Information

- **Setting Default Parameter Values and BDF Instance Parameter Values** on page 16-16
  For more information about the GUI-based entry methods, the interpretation of parameter values, and format recommendations
- **Passing Parameters Between Two Design Languages** on page 16-17
  For more information about parameter type rules

Setting Default Parameter Values and BDF Instance Parameter Values

Default parameter values and BDF instance parameter values do not have an explicitly declared type. Usually, the Quartus II software can correctly infer the type from the value without ambiguity. For example, the Quartus II software interprets “`ABC`” as a string, `123` as an integer, and `15.4` as a floating-point value. In other cases, such as when the instantiated subdesign language is VHDL, the Quartus II software uses the type of the parameter, generic, or both in the instantiated entity to determine how to interpret the value, so that the Quartus II software interprets a value of `123` as a string if the VHDL parameter is of a type string. In addition, you can set the parameter value in a format that is legal in the language of the instantiated entity. For example, to pass an unsized bit literal value from **.bdf** to Verilog HDL, you can use `'1` as the parameter value, and to pass a 4-bit binary vector from **.bdf** to Verilog HDL, you can use `4'b1111` as the parameter value.

In a few cases, the Quartus II software cannot infer the correct type of parameter value. To avoid ambiguity, specify the parameter value in a type-encoded format in which the first or first and second characters of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string `1001` from **.bdf** to Verilog HDL, you cannot use the value `1001`, because the Quartus II software interprets it as a decimal value. You also cannot use the string `"1001"` because the Quartus II software interprets it as an ASCII string. You must use the type-encoded string `B"1001"` for the Quartus II software to correctly interpret the parameter value.

This table lists valid parameter strings and how the Quartus II software interprets the parameter strings. Use the type-encoded format only when necessary to resolve ambiguity.

Table 16-4: Valid Parameter Strings and Interpretations

Parameter String	Quartus II Parameter Type, Format, and Value
`S"abc"`, `s"abc"`	String value abc
`"abc123"`, `"123abc"`	String value abc123 or 123abc
`F"12.3"`, `f"12.3"`	Floating point number 12.3
`-5.4`	Floating point number -5.4
`D"123"`, `d"123"`	Decimal number 123
`123`, `-123`	Decimal number 123, -123
`X"ff"`, `H"ff"`	Hexadecimal value FF

Parameter String	Quartus II Parameter Type, Format, and Value
`Q"77", O"77"`	Octal value 77
`B"1010", b"1010"`	Unsigned binary value 1010
`SB"1010", sb"1010"`	Signed binary value 1010
`R"1", R"0", R"X", R"Z", r"1", r"0", r"X", r"Z"`	Unsized bit literal
`E"apple", e"apple"`	Enumeration type, value name is apple
`P"1 unit"`	Physical literal, the value is (1, unit)
`A(...), a(...)`	Array type or record type. The string (...) determines the array type or record type content

You can select the parameter type for global parameters or global constants with the pull-down list in the **Parameter** tab of the **Symbol Properties** dialog box. If you do not specify the parameter type, the Quartus II software interprets the parameter value and defines the parameter type. You must specify parameter type with the pull-down list to avoid ambiguity.

Note: If you open a **.bdf** in the Quartus II software, the software automatically updates the parameter types of old symbol blocks by interpreting the parameter value based on the language-independent format. If the Quartus II software does not recognize the parameter value type, the software sets the parameter type as **untyped**.

The Quartus II software supports the following parameter types:

- **Unsigned Integer**
- **Signed Integer**
- **Unsigned Binary**
- **Signed Binary**
- **Octal**
- **Hexadecimal**
- **Float**
- **Enum**
- **String**
- **Boolean**
- **Char**
- **Untyped/Auto**

Passing Parameters Between Two Design Languages

When passing a parameter between two different languages, a design block that is higher in the design hierarchy instantiates a lower-level subdesign block and provides parameter information. The subdesign language (the design entity that you instantiate) must correctly interpret the parameter. Based on the information provided by the higher-level design and the value format, and sometimes by the parameter type of the subdesign entity, the Quartus II software interprets the type and value of the passed parameter.

When passing a parameter whose value is an enumerated type value or literal from a language that does not support enumerated types to one that does (for example, from Verilog HDL to VHDL), you must ensure that the enumeration literal is in the correct spelling in the language of the higher-level design block (block that is higher in the hierarchy). The Quartus II software passes the parameter value as a string literal, and the language of the lower-level design correctly convert the string literal into the correct enumeration literal.

If the language of the lower-level entity is SystemVerilog, you must ensure that the `enum` value is in the correct case. In SystemVerilog, two enumeration literals differ in more than just case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion and is more difficult to pass parameters from case-insensitive HDLs, such as VHDL.

Arrays have different support in different design languages. For details about the array parameter format, refer to the **Parameter** section in the Analysis & Synthesis Report of a design that contains array parameters or generics.

The following code shows examples of passing parameters from one design entry language to a subdesign written in another language.

Table 16-5: VHDL Parameterized Subdesign Entity

This table shows a VHDL subdesign that you instantiate in a top-level Verilog HDL design in Table 16-6.

HDL	Code

VHDL	`type fruit is (apple, orange, grape);`
	`entity vhdl_sub is`
	`generic (`
	`name : string := "default",`
	`width : integer := 8,`
	`number_string : string := "123",`
	`f : fruit := apple,`
	`binary_vector : std_logic_vector(3 downto 0) := "0101",`
	`signed_vector : signed (3 downto 0) := "1111");`

Table 16-6: Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity

This table shows a Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity from Table 16-5.

HDL	Code

Verilog HDL	`vhdl_sub inst (...);`
	`defparam inst.name = "lower";`
	`defparam inst.width = 3;`
	`defparam inst.num_string = "321";`
	`defparam inst.f = "grape"; // Must exactly match enum value`
	`defparam inst.binary_vector = 4'b1010;`
	`      defparam inst.signed_vector = 4'sb1010;`

Table 16-7: Verilog HDL Parameterized Subdesign Module

This table shows a Verilog HDL subdesign that you instantiate in a top-level VHDL design in **Table 16-8**.

HDL	Code

Verilog HDL	`module veri_sub (...)`
	`parameter name = "default";`
	`parameter width = 8;`
	`parameter number_string = "123";`
	`parameter binary_vector = 4'b0101;`
	`parameter signed_vector = 4'sb1111;`

Table 16-8: VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module

This table shows a VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from **Table 16-7**.

HDL	Code

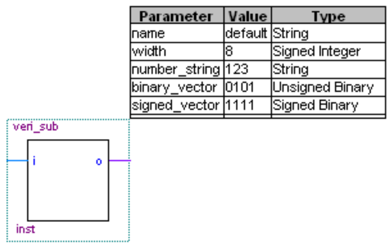
VHDL	`inst:veri_sub`
	`generic map (`
	`name => "lower",`
	`width => 3,`
	`number_string => "321"`
	`binary_vector = "1010"`
	`signed_vector = "1010")`

To use an HDL subdesign such as the one shown in **Table 16-7** in a top-level **.bdf** design, you must generate a symbol for the HDL file, as shown in **Figure 16-2**. Open the HDL file in the Quartus II software, and then, on the File menu, point to **Create/Update,** and then click **Create Symbol Files for Current File**.

To specify parameters on a **.bdf** instance, double-click the parameter value box for the instance symbol, or right-click the symbol and click **Properties**, and then click the **Parameters** tab. Right-click the symbol and click **Update Design File from Selected Block** to pass the updated parameter to the HDL file.

Figure 16-2: BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module

This figure shows BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from **Table 16-7**



Parameter	Value	Type
name	default	String
width	8	Signed Integer
number_string	123	String
binary_vector	0101	Unsigned Binary
signed_vector	1111	Signed Binary

Incremental Compilation

Incremental compilation manages a design hierarchy for incremental design by allowing you to divide your design into multiple partitions. Incremental compilation ensures that the Quartus II software resynthesizes only the updated partitions of your design during compilation, to reduce the compilation time and the runtime memory usage. The feature maintains node names during synthesis for all registered and combinational nodes in unchanged partitions. You can perform incremental synthesis by setting the netlist type for all design partitions to **Post-Synthesis**.

You can also preserve the placement and routing information for unchanged partitions. This feature allows you to preserve performance of unchanged blocks in your design and reduces the time required for placement and routing, which significantly reduces your design compilation time.

Related Information

- **About Incremental Compilation**
  For more information about incremental compilation
- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments**
  For more information about incremental compilation best practices and floorplan assignments
- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1
  For more information about incremental compilation for hierarchical and team-based design

Partitions for Preserving Hierarchical Boundaries

A design partition represents a portion of your design that you want to synthesize and fit incrementally.

If you want to preserve the **Optimization Technique** and **Restructure Multiplexers** logic options in any entity, you must create new partitions for the entity instead of using the **Preserve Hierarchical Boundary** logic option. If you have settings applied to specific existing design hierarchies, particularly those created in the Quartus II software versions before 9.0, you must create a design partition for the design hierarchy so that synthesis can optimize the design instance independently and preserve the hierarchical boundaries.

Note: The **Preserve Hierarchical Boundary** logic option is available only in Quartus II software versions 8.1 and earlier. Altera recommends using design partitions if you want to preserve hierarchical boundaries through the synthesis and fitting process, because incremental compilation maintains the hierarchical boundaries of design partitions.

Parallel Synthesis

The **Parallel Synthesis** logic option reduces compilation time for synthesis. The option enables the Quartus II software to use multiple processors to synthesize multiple partitions in parallel.

This option is available when you perform the following tasks:

- Specify the maximum number of processors allowed under **Parallel Compilation** options in the **Compilation Process Settings** page of the **Settings** dialog box.
- Enable the incremental compilation feature.
- Use two or more partitions in your design.
- Turn on the **Parallel Synthesis** option.

By default, the Quartus II software enables the **Parallel Synthesis** option. To disable parallel synthesis, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** > **Parallel Synthesis**.

You can also set the **Parallel Synthesis** option with the following Tcl command:

```
set_global_assignment -name parallel_synthesis off
```

If you use the command line, you can differentiate among the interleaved messages by turning on the **Show partition that generated the message** option in the Messages page. This option shows the partition ID in parenthesis for each message.

You can view all the interleaved messages from different partitions in the Messages window. The **Partition** column in the Messages window displays the partition ID of the partition referred to in the message. After compilation, you can sort the messages by partition.

Related Information

About the Messages Window
For more information about displaying the Partition column

Quartus II Exported Partition File as Source

You can use a **.qxp** as a source file in incremental compilation. The **.qxp** contains the precompiled design netlist exported as a partition from another Quartus II project, and fully defines the entity. Project team members or intellectual property (IP) providers can use a **.qxp** to send their design to the project lead, instead of sending the original HDL source code. The **.qxp** preserves the compilation results and instance-specific assignments. Not all global assignments can function in a different Quartus II project. You can override the assignments for the entity in the **.qxp** by applying assignments in the top-level design.

Related Information

Quartus II Exported Partition File .qxp
For more information about **.qxp**

Quartus II Incremental Compilation for Hierarchical and Team-Based Design on page 3-1
For more information about exporting design partitions and using **.qxp** files

Quartus II Synthesis Options

The Quartus II software offers several options to help you control the synthesis process and achieve optimal results for your design.

Note: When you apply a Quartus II Synthesis option globally or to an entity, the option affects all lower-level entities in the hierarchy path, including entities instantiated with Altera and third-party IP.

Related Information

- Setting Synthesis Options on page 16-22
  Describes the **Compiler Settings** page of the **Settings** dialog box, in which you can set the most common global settings and options, and defines the following types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives.

Setting Synthesis Options

You can set synthesis options in the **Settings** dialog box, or with logic options in the Quartus II software, or you can use synthesis attributes and directives in your HDL source code.

The **Compiler Settings** page of the **Settings** dialog box allows you to set global synthesis options that apply to the entire project. You can also use a corresponding Tcl command.

You can set some of the advanced synthesis settings in the **Advanced Settings** dialog box on the **Compiler Settings** page.

Related Information

Netlist Optimizations and Physical Synthesis
For more information about Physical Synthesis options

Quartus II Logic Options

The Quartus II logic options control many aspects of the synthesis and placement and routing process. To set logic options in the Quartus II software, on the Assignments menu, click **Assignment Editor**. You can also use a corresponding Tcl command to set global assignments. The Quartus II logic options enable you to set instance or node-specific assignments without editing the source HDL code.

Related Information

About the Assignment Editor
For more information about using the Assignment Editor

Synthesis Attributes

The Quartus II software supports synthesis attributes for Verilog HDL and VHDL, also commonly called pragmas. These attributes are not standard Verilog HDL or VHDL commands. Synthesis tools use attributes to control the synthesis process. The Quartus II software applies the attributes in the HDL source code, and attributes always apply to a specific design element. Some synthesis attributes are also available as Quartus II logic options via the Quartus II software or scripting. Each attribute description indicates a corresponding setting or a logic option that you can set in the Quartus II software. You can specify only some attributes with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the Assignment Editor or in the **.qsf**. Assignments or settings made with the Quartus II software, the **.qsf**, or the Tcl interface take precedence over assignments or settings made with synthesis attributes in your HDL code. The Quartus II software generates warning messages if the software finds invalid attributes, but does not generate an error or stop the compilation. This behavior is necessary because attributes are specific to various design tools, and attributes not recognized in the Quartus II software might be for a different EDA tool. The Quartus II software lists the attributes specified in your HDL code in the Source assignments table of the Analysis & Synthesis report.

The Verilog-2001, SystemVerilog, and VHDL language definitions provide specific syntax for specifying attributes, but in Verilog-1995, you must embed attribute assignments in comments. You can enter attributes in your code using the syntax in Specifying Synthesis Attributes in Verilog-1995 on page 1-23 through Synthesis Attributes in VHDL on page 1-24, in which <*attribute*>, <*attribute type*>, <*value*>, <*object*>, and <*object type*> are variables, and the entry in brackets is optional. These examples demonstrate each syntax form.

Note: Verilog HDL is case sensitive; therefore, synthesis attributes in Verilog HDL files are also case sensitive.

In addition to the `synthesis` keyword shown above, the Quartus II software supports the `pragma`, `synopsys`, and `exemplar` keywords for compatibility with other synthesis tools. The software also supports the `altera` keyword, which allows you to add synthesis attributes that the Quartus II Integrated Synthesis feature recognizes and not by other tools that recognize the same synthesis attribute.

Note: Because formal verification tools do not recognize the `exemplar`, `pragma`, and `altera` keywords, avoid using these attribute keywords when using formal verification.

Related Information

- **Maximum Fan-Out** on page 16-42
  For more information about maximum fan-out attribute
- **Preserve Registers** on page 16-38
  For more information about `preserve` attribute

Synthesis Attributes in Verilog-1995

You must use Verilog-1995 comment-embedded attributes as a suffix to the declaration of an item and must appear before a semicolon, when a semicolon is necessary.

Note: You cannot use the open one-line comment in Verilog HDL when a semicolon is necessary after the line, because it is not clear to which HDL element that the attribute applies. For example, you cannot make an attribute assignment such as `reg r; // synthesis <attribute>` because the Quartus II software could read the attribute as part of the next line.

Specifying Synthesis Attributes in Verilog-1995

The following show an example of specifying synthesis attributes in Verilog-1995:

```
// synthesis <attribute> [ = <value> ]
or
/* synthesis <attribute> [ = <value> ] */
```

Applying Multiple Attributes to the Same Instance in Verilog-1995

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces.

```
//synthesis <attribute1> [ = <value> ] <attribute2> [ = <value> ]
```

For example, to set the `maxfan` attribute to `16` and set the `preserve` attribute on a register called `my_reg`, use the following syntax:

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

Related Information

- **Maximum Fan-Out** on page 16-42
  For more information about maximum fan-out attribute
- **Preserve Registers** on page 16-38
  For more information about `preserve` attribute

Synthesis Attributes in Verilog-2001

You must use Verilog-2001 attributes as a prefix to a declaration, module item, statement, or port connection, and as a suffix to an operator or a Verilog HDL function name in an expression.

Note: Formal verification does not support the Verilog-2001 attribute syntax because the tools do not recognize the syntax.

Specifying Synthesis Attributes in Verilog-2001 and SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Applying Multiple Attributes

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas.

```
(* <attribute1> [ = <value1>], <attribute2> [ = <value2> ] *)
```

For example, to set the `maxfan` attribute to `16` and set the `preserve` attribute on a register called `my_reg`, use the following syntax:

```
(* maxfan = 16, preserve *) reg my_reg;
```

Related Information

- **Maximum Fan-Out** on page 16-42
  For more information about maximum fan-out attribute
- **Preserve Registers** on page 16-38
  For more information about `preserve` attribute

Synthesis Attributes in VHDL

VHDL attributes declare and apply the attribute type to the object you specify.

Synthesis Attributes in VHDL

The following shows the synthesis attributes example in VHDL:

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

altera_syn_attributes

The Quartus II software defines and applies each attribute separately to a given node. For VHDL designs, the software declares all supported synthesis attributes in the `altera_syn_attributes` package in the Altera library. You can call this library from your VHDL code to declare the synthesis attributes:

```
LIBRARY altera;
USE altera.altera_syn_attributes.all;
```

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called compiler directives or pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not standard Verilog HDL or VHDL commands. Synthesis tools use directives to control the synthesis process. Directives do not apply to a specific design node, but change the behavior of the synthesis tool from the point in which they occur in the HDL source code. Other tools, such as simulators, ignore these directives and treat them as comments.

Table 16-9: Specifying Synthesis Directives

You can enter synthesis directives in your code using the syntax in the following table, in which <*directive*> and <*value*> are variables, and the entry in brackets are optional. For synthesis directives, no equal sign before the value is necessary; this is different than the Verilog syntax for synthesis attributes. The examples demonstrate each syntax form.

Language	Syntax Example
Verilog HDL[10]	`// synthesis <directive> [ <value> ]` or `/* synthesis <directive> [ <value> ] */`
VHDL	`-- synthesis <directive> [ <value> ]`
VHDL-2008	`/* synthesis <directive> [<value>] */`

In addition to the `synthesis` keyword shown above, the software supports the `pragma`, `synopsys`, and `exemplar` keywords in Verilog HDL and VHDL for compatibility with other synthesis tools. The Quartus II software also supports the keyword `altera`, which allows you to add synthesis directives that only Quartus II Integrated Synthesis feature recognizes, and not by other tools that recognize the same synthesis directives.

Note: Because formal verification tools ignore the `exemplar`, `pragma`, and `altera` keywords, Altera recommends that you avoid using these directive keywords when you use formal verification to prevent mismatches with the Quartus II results.

Optimization Technique

The **Optimization Technique** logic option specifies the goal for logic optimization during compilation; that is, whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two.

Related Information

Optimization Technique logic option
For more information about the Optimization Technique logic option

[10] Verilog HDL is case sensitive; therefore, all synthesis directives are also case sensitive.

Auto Gated Clock Conversion

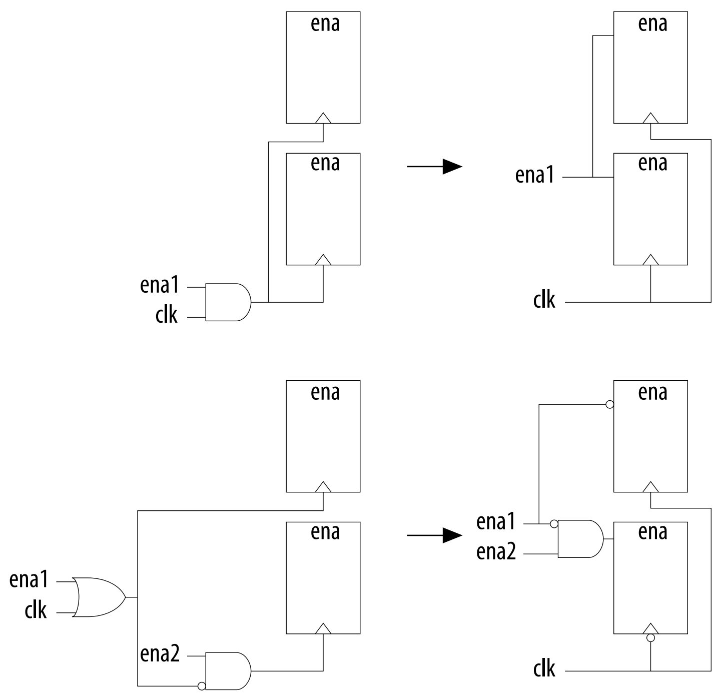
Clock gating is a common optimization technique in ASIC designs to minimize power consumption. You can use the **Auto Gated Clock Conversion** logic option to optimize your prototype ASIC designs by converting gated clocks into clock enables when you use FPGAs in your ASIC prototyping. The automatic conversion of gated clocks to clock enables is more efficient than manually modifying source code. The **Auto Gated Clock Conversion** logic option automatically converts qualified gated clocks (base clocks as defined in the Synopsys Design Constraints [SDC]) to clock enables. Click **AssignmentsSettingsCompiler SettingsAdvanced Settings (Synthesis)** to enable **Auto Gated Clock Conversion**.

The gated clock conversion occurs when all these conditions are met:

- Only one base clock drives a gated-clock
- For one set of gating input values, the value output of the gated clock remains constant and does not change as the base clock changes
- For one value of the base clock, changes in the gating inputs do not change the value output for the gated clock

The option supports combinational gates in clock gating network.

Figure 16-3: Example Gated Clock Conversion



Note: This option does not support registers in RAM, DSP blocks, or I/O related WYSIWYG primitives. Because the gated-clock conversion cannot trace the base clock from the gated clock, the gated clock conversion does not support multiple design partitions from incremental compilation in which the gated clock and base clock are not in the same hierarchical partition. A gated clock tree, instead of every gated clock, is the basis of each conversion. Therefore, if you cannot convert a gated clock from a root gated clock of a multiple cascaded gated clock, the conversion of the entire gated clock tree fails.

The **Info** tab in the Messages window lists all the converted gated clocks. You can view a list of converted and nonconverted gated clocks from the Compilation Report under the **Optimization Results** of the Analysis & Synthesis Report. The **Gated Clock Conversion Details** table lists the reasons for nonconverted gated clocks.

Related Information

Auto Gated Clock Conversion logic option

For more information about Auto Gated Clock Conversion logic option and a list of supported devices

Timing-Driven Synthesis

The **Timing-Driven Synthesis** logic option specifies whether Analysis & Synthesis should use the SDC timing constraints of your design to better optimize the circuit. When you turn on this option, Analysis & Synthesis runs timing analysis to obtain timing information about the netlist, and then considers the SDC timing constraints to focus on critical portions of your design when optimizing for performance, while optimizing noncritical portions for area. When you turn on this option, Analysis & Synthesis also protects SDC constraints by not merging duplicate registers that have incompatible timing constraints.

When you turn on the **Timing-Driven Synthesis** logic option, Analysis & Synthesis increases performance by improving logic depth on critical portions of your design, and improving area on noncritical portions of your design. The increased performance affects the amount of area used, specifically adaptive look-up tables (ALUTs) and registers in your design. Depending on how much of your design is timing critical, overall area can increase or decrease when you turn on the **Timing-Driven Synthesis** logic option. Runtime and peak memory use increases slightly if you turn on the **Timing-Driven Synthesis** logic option.

When you turn on the **Timing-Driven Synthesis** logic option, the **Optimization Technique** logic option has the following effect. With **Optimization Technique Speed**, Timing-Driven Synthesis optimizes timing-critical portions of your design for performance at the cost of increasing area (logic and register utilization). With an **Optimization Technique** of **Balanced**, Timing-Driven Synthesis also optimizes the timing-critical portions of your design for performance, but the option allows only limited area increase. With **Optimization Technique Area**, Timing-Driven Synthesis optimizes your design only for area. **Timing-Driven Synthesis** prevents registers with incompatible timing constraints from merging for any **Optimization Technique** setting. If your design contains multiple partitions, you can select **Timing-Driven Synthesis** unique options for each partition. If you use a **.qxp** as a source file, or if your design uses partitions developed in separate Quartus II projects, the software cannot properly compute timing of paths that cross the partition boundaries.

Even with the **Optimization Technique** logic option set to **Speed**, the **Timing-Driven Synthesis** option still considers the resource usage in your design when increasing area to improve timing. For example, the **Timing-Driven Synthesis** option checks if a device has enough registers before deciding to implement the shift registers in logic cells instead of RAM for better timing performance.

When using incremental compilation, Integrated Synthesis allows each partition to use up all the registers in a device. You can use the **Maximum Number of LABs** settings to specify the number of LABs that every partition can use. If your design has only one partition, you can also use the **Maximum Number of LABs** settings to limit the number of resources that your design can use. This limitation is useful when you add more logic to your design.

To turn on or turn off the **Timing-Driven Synthesis** logic option, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.
2. Turn on or turn off **Timing-Driven Synthesis**.

Note: Altera recommends that you select a specific device for timing-driven synthesis to have the most accurate timing information. When you select auto device, timing-driven synthesis uses the smallest device for the selected family to obtain timing information.

Related Information

Timing-Driven Synthesis logic option
For more information about Timing-Driven Synthesis logic option and a list of supported devices

SDC Constraint Protection on page 16-28
For more information about SDC constraint protection

SDC Constraint Protection

The **SDC Constraint Protection** option specifies whether Analysis & Synthesis should protect registers from merging when they have incompatible timing constraints. For example, when you turn on this option, the software does not merge two registers that are duplicates of each other but have different multicycle constraints on them. When you turn on the **Timing-Driven Synthesis** option, the software detects registers with incompatible constraints, and you do not need to turn on **SDC Constraint Protection**. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** to enable the **SDC constraint protection** option.

PowerPlay Power Optimization

The **PowerPlay Power Optimization** logic option controls the power-driven compilation setting of Analysis & Synthesis and determines how aggressively Analysis & Synthesis optimizes your design for power.

Related Information

- **PowerPlay Power Optimization logic option**
  For more information about the available settings for the PowerPlay power optimization logic option and a list of supported devices
- **Power Optimization**
  For more information about optimizing your design for power utilization
- **PowerPlay Power Analysis**
  For information about analyzing your power results

Limiting Resource Usage in Partitions

Resource balancing is important when performing Analysis & Synthesis. During resource balancing, Quartus II Integrated Synthesis considers the amount of used and available DSP and RAM blocks in the device, and tries to balance these resources to prevent no-fit errors.

For DSP blocks, Resource balancing is important when performing Analysis & Synthesis. During resource balancing, Quartus II Integrated Synthesis considers the amount of used and available DSP and RAM blocks in the device, and tries to balance these resources to prevent no-fit errors. resource balancing converts the remaining DSP blocks to equivalent logic if there are more DSP blocks in your design that the software can place in the device. For RAM blocks, resource balancing converts RAM blocks to different types of RAM blocks if there are not enough blocks of a certain type available in the device; however, Quartus II Integrated Synthesis does not convert RAM blocks to logic.

Note: The RAM balancing feature does not support Stratix V devices because Stratix V has only M20K memory blocks.

By default, Quartus II Integrated Synthesis considers the information in the targeted device to identify the number of available DSP or RAM blocks. However, in incremental compilation, each partition considers the information in the device independently and consequently assumes that the partition has all the DSP and RAM blocks in the device available for use, resulting in over allocation of DSP or RAM blocks in your design, which means that the total number of DSP or RAM blocks used by all the partitions is greater than the number of DSP or RAM blocks available in the device, leading to a no-fit error during the fitting process.

Related Information

- **Creating LogicLock Regions** on page 16-29
  For more information about preventing a no-fit error during the fitting process
- **Using Assignments to Limit the Number of RAM and DSP Blocks** on page 16-29
  For more information about preventing a no-fit error during the fitting process

Creating LogicLock Regions

The floorplan-aware synthesis feature allows you to use LogicLock regions to define resource allocation for DSP blocks and RAM blocks. For example, if you assign a certain partition to a certain LogicLock region, resource balancing takes into account that all the DSP and RAM blocks in that partition need to fit in this LogicLock region. Resource balancing then balances the DSP and RAM blocks accordingly.

Because floorplan-aware balancing step considers only one partition at a time, it does not know that nodes from another partition may be using the same resources. When using this feature, Altera recommends that you do not manually assign nodes from different partitions to the same LogicLock region.

If you do not want the software to consider the LogicLock floorplan constraints when performing DSP and RAM balancing, you can turn off the floorplan-aware synthesis feature. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** to disable **Use LogicLock Constraints During Resource Balancing** option.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design** on page 3-1
  For more information about using LogicLock regions to create a floorplan for incremental compilation

Using Assignments to Limit the Number of RAM and DSP Blocks

For DSP and RAM block balancing, you can use assignments to limit the maximum number of blocks that the balancer allows. You can set these assignments globally or on individual partitions. For DSP block balancing, the **Maximum DSP Block Usage** logic option allows you to specify the maximum number of DSP blocks that the DSP block balancer assumes are available for the current partition. For RAM blocks, the floorplan-aware logic option allows you to specify maximum resources for different RAM types, such as **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks**, **Maximum Number of M512 Memory Blocks**, **Maximum Number of M-RAM/M144K Memory Blocks**, or **Maximum Number of LABs**.

The partition-specific assignment overrides the global assignment, if any. However, each partition that does not have a partition-specific assignment uses the value set by the global assignment, or the value derived from the device size if no global assignment exists. This action can also lead to over allocation. Therefore, Altera recommends that you always set the assignment on each partition individually.

To select the **Maximum Number <block type> Memory Blocks** option or the **Maximum DSP Block Usage** option globally, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**. You can use the Assignment Editor to set this assignment on a partition by selecting the assignment, and setting it on the root entity of a partition. You can set any positive integer as the value of this assignment. If you set this assignment on a name other than a partition root, Analysis & Synthesis gives an error.

Related Information

- **Maximum DSP Block Usage logic option**
  For more information about the **Maximum DSP Block Usage** logic option, including a list of supported device families
- **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks logic option**
  For more information about the **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks** logic option, including a list of supported device families
- **Maximum Number of M512 Memory Blocks logic option**
  For more information about the **Maximum Number of M512 Memory Blocks** logic option, including a list of supported device families
- **Maximum Number of M-RAM/144K Memory Blocks logic option**
  For more information about **Maximum Number of M-RAM/144K Memory Blocks** logic option, including a list of supported device families
- **Maximum Number of LABs logic option**
  For more information about the **Maximum Number of LABs** logic option, including a list of supported device families

Restructure Multiplexers

The **Restructure Multiplexers logic** option restructures multiplexers to create more efficient use of area, allowing you to implement multiplexers with a reduced number of LEs or ALMs.

When multiplexers from one part of your design feed multiplexers in another part of your design, trees of multiplexers form. Multiplexers may arise in different parts of your design through Verilog HDL or VHDL constructs such as the "`if`," "`case`," or "`?:`" statements. Multiplexer buses occur most often as a result of multiplexing together arrays in Verilog HDL, or `STD_LOGIC_VECTOR` signals in VHDL. The **Restructure Multiplexers** logic option identifies buses of multiplexer trees that have a similar structure. This logic option optimizes the structure of each multiplexer bus for the target device to reduce the overall amount of logic in your design.

Results of the multiplexer optimizations are design dependent, but area reductions as high as 20% are possible. The option can negatively affect your design's f_{MAX}.

Related Information

- **Recommended HDL Coding Styles** on page 12-1
  For more information about optimizing for multiplexers

- **Analysis Synthesis Optimization Results Reports**
  For more information about the Multiplexer Restructuring Statistics report table for each bus of multiplexers
- **Restructure Multiplexers logic option**
  For more information about the Restructure Multiplexers logic option, including the settings and a list of supported device families

Synthesis Effort

The **Synthesis Effort** logic option specifies the overall synthesis effort level in the Quartus II software.

Related Information

Synthesis Effort logic option
For more information about Synthesis Effort logic option, including a list of supported device families

Fitter Intial Placement Seed

The **Fitter Intial Placement Seed** option specifies the seed that Synthesis uses to randomly run synthesis in a slightly different way. You can use this seed when your design is close to meeting requirements, to get a slightly different result. The seeds that produce the best result for a design might change if your design changes.

To set the **Synthesis Seed** option, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**. The default value is **1**. You can specify a positive integer value.

State Machine Processing

The **State Machine Processing** logic option specifies the processing style to synthesize a state machine.

The default state machine encoding, **Auto**, uses one-hot encoding for FPGA devices and minimal-bits encoding for CPLDs. These settings achieve the best results on average, but another encoding style might be more appropriate for your design, so this option allows you to control the state machine encoding.

For one-hot encoding, the Quartus II software does not guarantee that each state has one bit set to one and all other bits set to zero. Quartus II Integrated Synthesis creates one-hot register encoding with standard one-hot encoding and then inverts the first bit. This results in an initial state with all zero values, and the remaining states have two 1 values. Quartus II Integrated Synthesis encodes the initial state with all zeros for the state machine power-up because all device registers power up to a low value. This encoding has the same properties as true one-hot encoding: the software recognizes each state by the value of one bit. For example, in a one-hot-encoded state machine with five states, including an initial or reset state, the software uses the following register encoding:

```
State 0    0 0 0 0 0
State 1    0 0 0 1 1
State 2    0 0 1 0 1
State 3    0 1 0 0 1
State 4    1 0 0 0 1
```

If you set the **State Machine Processing** logic option to **User-Encoded** in a Verilog HDL design, the software starts with the original design values for the state constants. For example, a Verilog HDL design can contain the following declaration:

```
parameter S0 = 4'b1010, S1 = 4'b0101, ...
```

If the software infers the states `S0, S1,...` the software uses the encoding `4'b1010, 4'b0101,...`. If necessary, the software inverts bits in a user-encoded state machine to ensure that all bits of the reset state of the state machine are zero.

Note: You can view the state machine encoding from the Compilation Report under the State Machines of the Analysis & Synthesis Report. The State Machine Viewer displays only a graphical representation of the state machines as interpreted from your design.

To assign your own state encoding with the **User-Encoded** setting of the **State Machine Processing** option in a VHDL design, you must apply specific binary encoding to the elements of an enumerated type because enumeration literals have no numeric values in VHDL. Use the `syn_encoding` synthesis attribute to apply your encoding values.

Related Information

- **Manually Specifying State Assignments Using the syn_encoding Attribute** on page 16-32
- **Recommended HDL Coding Styles** on page 12-1
  For guidelines on how to correctly infer and encode your state machine
- **Analyzing Designs with Quartus II Netlist Viewers**
  For more information about the State Machine Viewer
- **State Machine Processing logic option**
  For information about the State Machine Processing logic option, including the settings and supported devices
- **Manually Specifying Enumerated Types Using the enum_encoding Attribute** on page 16-33
  For more information about assigning your own state encoding with the **User-Encoded** setting of the **State Machine Processing** option in a VHDL design

Manually Specifying State Assignments Using the syn_encoding Attribute

The Quartus II software infers state machines from enumerated types and automatically assigns state encoding based on **State Machine Processing** on page 16-31.

With this logic option, you can choose the value **User-Encoded** to use the encoding from your HDL code. However, in standard VHDL code, you cannot specify user encoding in the state machine description because enumeration literals have no numeric values in VHDL.

To assign your own state encoding for the **User-Encoded State Machine Processing** setting, use the `syn_encoding` synthesis attribute to apply specific binary encodings to the elements of an enumerated type or to specify an encoding style. The Quartus II software can implement Enumeration Types with different encoding styles, as listed in this table.

Table 16-10: syn_encoding Attribute Values

Attribute Value	Enumeration Types
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If the number of literals is less than five, use the `"sequential"` encoding. If the number of literals is more than five, but fewer than 50, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding 0 and the second 1.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2^N states, but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring N bits, in which N is the number of enumeration literals in the Enumeration Type.
`"compact"`	Use an encoding with the fewest bits.
`"user"`	Encode each state using its value in the Verilog source. By changing the values of your state constants, you can change the encoding of your state machine.

The `syn_encoding` attribute must follow the enumeration type definition, but precede its use.

Related Information

State Machine Processing on page 16-31

Manually Specifying Enumerated Types Using the enum_encoding Attribute

By default, the Quartus II software one-hot encodes all enumerated types you defined. With the `enum_encoding` attribute, you can specify the logic encoding for an enumerated type and override the default `one-hot` encoding to improve the logic efficiency.

Note: If an enumerated type represents the states of a state machine, using the `enum_encoding` attribute to specify a manual state encoding prevents the Compiler from recognizing state machines based on the enumerated type. Instead, the Compiler processes these state machines as regular logic with the encoding specified by the attribute, and the Report window for your project does not list these states machines as state machines. If you want to control the encoding for a recognized state machine, use the **State Machine Processing** logic option and the `syn_encoding` synthesis attribute.

To use the `enum_encoding` attribute in a VHDL design file, associate the attribute with the enumeration type whose encoding you want to control. The `enum_encoding` attribute must follow the enumeration type definition, but precede its use. In addition, the attribute value should be a string literal that specifies either an arbitrary user encoding or an encoding style of `"default"`, `"sequential"`, `"gray"`, `"johnson"`, or `"one-hot"`.

An arbitrary user encoding consists of a space-delimited list of encodings. The list must contain as many encodings as the number of enumeration literals in your enumeration type. In addition, the encodings should have the same length, and each encoding must consist solely of values from the `std_ulogic` type declared by the `std_logic_1164` package in the IEEE library.

In this example, the `enum_encoding` attribute specifies an arbitrary user encoding for the enumeration type `fruit`.

Example 16-5: Specifying an Arbitrary User Encoding for Enumerated Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "11 01 10 00";
```

This example shows the encoded enumeration literals:

Example 16-6: Encoded Enumeration Literals

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

Altera recommends that you specify an encoding style, rather than a manual user encoding, especially when the enumeration type has a large number of enumeration literals. The Quartus II software can implement Enumeration Types with the different encoding styles, as shown in this table.

Table 16-11: enum_encoding Attribute Values

Attribute Value	Enumeration Types
`"default"`	Use an encoding based on the number of enumeration literals in the enumeration type. If the number of literals are fewer than five, use the `"sequential"` encoding. If the number of literals are more than five, but fewer than 50 literals, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the enumeration type has encoding `0` and the second `1`.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2^N states, but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring N bits, in which N is the number of enumeration literals in the enumeration type.

In **Example 16-5**, the `enum_encoding` attribute manually specified a gray encoding for the enumeration type `fruit`. You can also concisely write this example by specifying the `"gray"` encoding style instead of a manual encoding, as shown in the following example:

Example 16-7: Specifying the "gray" Encoding Style or Enumeration Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "gray";
```

Safe State Machine

The **Safe State Machine** logic option and corresponding `syn_encoding` attribute value `safe` specify that the software must insert extra logic to detect an illegal state, and force the transition of the state machine to the reset state.

A finite state machine can enter an illegal state—meaning the state registers contain a value that does not correspond to any defined state. By default, the behavior of the state machine that enters an illegal state is undefined. However, you can set the `syn_encoding` attribute to `safe` or use the **Safe State Machine** logic option if you want the state machine to recover deterministically from an illegal state. The software inserts extra logic to detect an illegal state, and forces the transition of the state machine to the reset state. You can use this logic option when the state machine enters an illegal state. The most common cause of an illegal state is a state machine that has control inputs that come from another clock domain, such as the control logic for a clock-crossing FIFO, because the state machine must have inputs from another clock domain. This option protects only state machines (and not other registers) by forcing them into the reset state. You can use this option if your design has asynchronous inputs. However, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

The `safe` state machine value does not use any user-defined default logic from your HDL code that corresponds to unreachable states. Verilog HDL and VHDL enable you to specify a behavior for all states in the state machine explicitly, including unreachable states. However, synthesis tools detect if state machine logic is unreachable and minimize or remove the logic. Synthesis tools also remove any flag signals or logic that indicate such an illegal state. If the software implements the state machine as safe, the recovery logic added by Quartus II Integrated Synthesis forces its transition from an illegal state to the reset state.

You can set the **Safe State Machine** logic option globally, or on individual state machines. To set this logic option, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

Table 16-12: Setting the `syn_encoding safe` attribute on a State Machine in HDL

HDL	Code
Verilog HDL	`reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */ ;`
Verilog-2001 and SystemVerilog	`(* syn_encoding = "safe" *) reg [2:0] my_fsm;`
VHDL	`ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";`

If you specify an encoding style, separate the encoding style value in the quotation marks with the `safe` value with a comma, as follows: `"safe, one-hot"` or `"safe, gray"`.

Safe state machine implementation can result in a noticeable area increase for your design. Therefore, Altera recommends that you set this option only on the critical state machines in your design in which the safe mode is necessary, such as a state machine that uses inputs from asynchronous clock domains. You may not need to use this option if you correctly synchronize inputs coming from other clock domains.

Note: If you create the `safe` state machine assignment on an instance that the software fails to recognize as a state machine, or an entity that contains a state machine, the software takes no action. You must restructure the code, so that the software recognizes and infers the instance as a state machine.

Related Information

- **Manually Specifying State Assignments Using the syn_encoding Attribute** on page 16-32
- **Safe State Machine logic option**
  For more information about the Safe State Machine logic option
- **Recommended HDL Coding Styles** on page 12-1
  For guidelines to ensure that the software correctly infers your state machine

Power-Up Level

This logic option causes a register (flipflop) to power up with the specified logic level, either high (`1`) or low (`0`). The registers in the core hardware power up to `0` in all Altera devices. For the register to power up with a logic level high, the Compiler performs an optimization referred to as NOT-gate push back on the register. NOT-gate push back adds an inverter to the input and the output of the register, so that the reset and power-up conditions appear to be high and the device operates as expected. The register itself still powers up to `0`, but the register output inverts so the signal arriving at all destinations is `1`.

The **Power-Up Level** option supports wildcard characters, and you can apply this option to any register, registered logic cell WYSIWYG primitive, or to a design entity containing registers, if you want to set the power level for all registers in your design entity. If you assign this option to a registered logic cell WYSIWYG primitive, such as an atom primitive from a third-party synthesis tool, you must turn on the **Perform WYSIWYG Primitive Resynthesis** logic option for the option to take effect. You can also apply the option to a pin with the logic configurations described in the following list:

- If you turn on this option for an input pin, the option transfers to the register that the pin drives, if all these conditions are present:
  - No logic, other than inversion, between the pin and the register.
  - The input pin drives the data input of the register.
  - The input pin does not fan-out to any other logic.
- If you turn on this option for an output or bidirectional pin, the option transfers to the register that feeds the pin, if all these conditions are present:
  - No logic, other than inversion, between the register and the pin.
  - The register does not fan out to any other logic.

Related Information

Power-Up Level logic option
For more information about the Power-Up Level logic option, including information on the supported device families

Inferred Power-Up Levels

Quartus II Integrated Synthesis reads default values for registered signals defined in Verilog HDL and VHDL code, and converts the default values into **Power-Up Level** settings. The software also synthesizes variables with assigned values in Verilog HDL initial blocks into power-up conditions. Synthesis of these default and initial constructs allows synthesized behavior of your design to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

The following register declarations all set a power-up level of V_{CC} or a logic value “1”, as shown in this example:

```
signal q : std_logic = '1';  -- power-up to VCC

reg q = 1'b1;  // power-up to VCC

reg q;
initial begin q = 1'b1; end  // power-up to VCC
```

Related Information

- Recommended HDL Coding Styles on page 12-1
  For more information about NOT-gate push back, the power-up states for Altera devices, and how set and reset control signals affect the power-up level

Power-Up Don’t Care

This logic option allows the Compiler to optimize registers in your design that do not have a defined power-up condition.

For example, your design might have a register with its D input tied to V_{CC}, and with no clear signal or other secondary signals. If you turn on this option, the Compiler can choose for the register to power up to V_{CC}. Therefore, the output of the register is always V_{CC}. The Compiler can remove the register and connect its output to V_{CC}. If you turn this option off or if you set a **Power-Up Level** assignment of **Low** for this register, the register transitions from GND to V_{CC} when your design starts up on the first clock signal. Thus, the register is at V_{CC} and you cannot remove the register. Similarly, if the register has a clear signal, the Compiler cannot remove the register because after asserting the clear signal, the register transitions again to GND and back to V_{CC}.

If the Compiler performs a **Power-Up Don’t Care** optimization that allows it to remove a register, it issues a message to indicate that it is doing so.

This project-wide option does not apply to registers that have the **Power-Up Level** logic option set to either **High** or **Low**.

Related Information

Power-Up Don’t Care logic option
For more information about Power-Up Don’t Care logic option and a list of supported devices

Remove Duplicate Registers

The **Remove Duplicate Registers** logic option removes registers that are identical to other registers.

Related Information

Remove Duplicate Registers logic option
For more information about Remove Duplicate Registers logic option and the supported devices

Preserve Registers

This attribute and logic option directs the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Optimizations can eliminate redundant registers and registers with constant drivers; this option prevents the software from reducing a register to a constant or merging with a duplicate register. This option can preserve a register so you can observe the register during simulation or with the SignalTap® II Logic Analyzer. Additionally, this option can preserve registers if you create a preliminary version of your design in which you have not specified the secondary signals. You can also use the attribute to preserve a duplicate of an I/O register so that you can place one copy of the I/O register in an I/O cell and the second in the core.

Note: This option cannot preserve registers that have no fan-out.

The **Preserve Registers** logic option prevents the software from inferring a register as a state machine.

You can set the **Preserve Registers** logic option in the Quartus II software, or you can set the `preserve` attribute in your HDL code. In these examples, the Quartus II software preserves the `my_reg` register.

Table 16-13: Setting the `syn_preserve` attribute in HDL Code

HDL	Code[11]
Verilog HDL	`reg my_reg /* synthesis syn_preserve = 1 */;`
Verilog-2001	`(* syn_preserve = 1 *) reg my_reg;`

Table 16-14: Setting the `preserve` attribute in HDL Code

In addition to `preserve`, the Quartus II software supports the `syn_preserve` attribute name for compatibility with other synthesis tools.

HDL	Code

VHDL	`signal my_reg : stdlogic;`
	`attribute preserve : boolean;`
	`attribute preserve of my_reg : signal is true;`

Related Information

Preserve Registers logic option
For more information about the Preserve Registers logic option and the supported devices

[11] The `= 1` after the `preserve` are optional, because the assignment uses a default value of `1` when you specify the assignment.

Noprune Synthesis Attribute/Preserve Fan-out Free Register Node on page 16-39
For more information about preventing the removal of registers with no fan-out

Disable Register Merging/Don’t Merge Register

This logic option and attribute prevents the specified register from merging with other registers and prevents other registers from merging with the specified register. When applied to a design entity, it applies to all registers in the entity.

You can set the **Disable Register Merging** logic option in the Quartus II software, or you can set the `dont_merge` attribute in your HDL code, as shown in these examples. In these examples, the logic option or the attribute prevents the `my_reg` register from merging.

Table 16-15: Setting the `dont_merge` attribute in HDL code

HDL	Code
Verilog HD	`reg my_reg /* synthesis dont_merge */;`
Verilog-2001 and SystemVer-ilog	`(* dont_merge *) reg my_reg;`

VHDL	`signal my_reg : stdlogic;`
	`attribute dont_merge : boolean;`
	`attribute dont_merge of my_reg : signal is true;`

Related Information

Disable Register Merging logic option
For more information about the **Disable Register Merging** logic option and the supported devices

Noprune Synthesis Attribute/Preserve Fan-out Free Register Node

This synthesis attribute and corresponding logic option direct the Compiler to preserve a fan-out-free register through the entire compilation flow. This option is different from the **Preserve Registers** option, which prevents the Quartus II software from reducing a register to a constant or merging with a duplicate register. Standard synthesis optimizations remove nodes that do not directly or indirectly feed a top-level output pin. This option can retain a register so you can observe the register in the Simulator or the SignalTap II Logic Analyzer. Additionally, this option can retain registers if you create a preliminary version of your design in which you have not specified the fan-out logic of the register.

You can set the **Preserve Fan-out Free Register Node** logic option in the Quartus II software, or you can set the `noprune` attribute in your HDL code, as shown in these examples. In these examples, the logic option or the attribute preserves the `my_reg` register.

Note: You must use the `noprune` attribute instead of the logic option if the register has no immediate fan-out in its module or entity. If you do not use the synthesis attribute, the software removes (or “prunes”) registers with no fan-out during Analysis & Elaboration before the logic synthesis stage applies any logic options. If the register has no fan-out in the full design, but has fan-out in its module or entity, you can use the logic option to retain the register through compilation.

The software supports the attribute name `syn_noprune` for compatibility with other synthesis tools.

Table 16-16: Setting the `noprune` attribute in HDL code

HDL	Code
Verilog HD	`reg my_reg /* synthesis syn_noprune */;`
Verilog-2001 and SystemVerilog	`(* noprune *) reg my_reg;`

VHDL	`signal my_reg : stdlogic;`
	`attribute noprune: boolean;`
	`attribute noprune of my_reg : signal is true;`

Related Information

Preserve Fan-out Free Register logic option

For more information about **Preserve Fan-out Free Register Node** logic option and a list of supported devices

Keep Combinational Node/Implement as Output of Logic Cell

This synthesis attribute and corresponding logic option direct the Compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. A wire that has a `keep` attribute or a node that has the **Implement as Output of Logic Cell** logic option applied becomes the output of a logic cell in the final synthesis netlist, and the name of the logic cell remains the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II Logic Analyzer.

Note: The option cannot keep nodes that have no fan-out. You cannot maintain node names for wires with tri-state drivers, or if the signal feeds a top-level pin of the same name (the software changes the node name to a name such as <net name>~`buf0`).

You can use the **Ignore LCELL Buffers** logic option to direct Analysis & Synthesis to ignore logic cell buffers that the **Implement as Output of Logic Cell** logic option or the `LCELL` primitive created. If you apply this logic option to an entity, it affects all lower-level entities in the hierarchy path.

Note: To avoid unintended design optimizations, ensure that any entity instantiated with Altera or third-party IP that relies on logic cell buffers for correct behavior does not inherit the **Ignore LCELL Buffers** logic option. For example, if an IP core uses logic cell buffers to manage high fan-out signals and inherits the **Ignore LCELL Buffers** logic option, the target device may no longer function properly.

You can turn off the **Ignore LCELL Buffers** logic option for a specific entity to override any assignments inherited from higher-level entities in the hierarchy path if logic cell buffers created by the **Implement as Output of Logic Cell** logic option or the `LCELL` primitive are required for correct behavior.

You can set the **Implement as Output of Logic Cell** logic option in the Quartus II software, or you can set the `keep` attribute in your HDL code, as shown in these tables. In these tables, the Compiler maintains the node name `my_wire`.

Table 16-17: Setting the keep Attribute in HDL code

HDL	Code
Verilog HD	`wire my_wire /* synthesis keep = 1 */;`
Verilog-2001	`(* keep = 1 *) wire my_wire;`

Table 16-18: Setting the syn_keep Attribute in HDL Code

In addition to keep, the Quartus II software supports the syn_keep attribute name for compatibility with other synthesis tools.

HDL	Code

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

Related Information

Implement as Output of Logic Cell logic option

For more information about the **Implement as Output of Logic Cell** logic option and the supported devices

Disabling Synthesis Netlist Optimizations with dont_retime Attribute

This attribute disables synthesis retiming optimizations on the register you specify. When applied to a design entity, it applies to all registers in the entity.

You can turn off retiming optimizations with this option and prevent node name changes, so that the Compiler can correctly use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II software to disable retiming along with other synthesis netlist optimizations, or you can set the dont_retime attribute in your HDL code, as shown in the following table. In the following table, the code prevents my_reg register from being retimed.

Table 16-19: Setting the dont_retime Attribute in HDL Code

HDL	Code
Verilog HDL	`reg my_reg /* synthesis dont_retime */;`
Verilog-2001 and SystemVerilo	`(* dont_retime *) reg my_reg;`

VHD	`signal my_reg : std_logic;`
	`attribute dont_retime : boolean;`
	`attribute dont_retime of my_reg :`
	`signal is true;`

Note: For compatibility with third-party synthesis tools, Quartus II Integrated Synthesis also supports the attribute `syn_allow_retiming`. To disable retiming, set `syn_allow_retiming` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when you set the attribute to `1` or `true`.

Disabling Synthesis Netlist Optimizations with dont_replicate Attribute

This attribute disables synthesis replication optimizations on the register you specify. When applied to a design entity, it applies to all registers in the entity.

You can turn off register replication (or duplication) optimizations with this option, so that the Compiler uses your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II software to disable replication along with other synthesis netlist optimizations, or you can set the `dont_replicate` attribute in your HDL code, as shown in these examples. In these examples, the code prevents the replication of the `my_reg` register.

Table 16-20: Setting the `dont_replicate` attribute in HDL Code

HDL	Code
Verilog HD	`reg my_reg /* synthesis dont_replicate  */;`
Verilog-2001 and SystemVerilog	`(* dont_replicate *) reg my_reg;`

VHDL	`signal my_reg : std_logic;`
	`attribute dont_replicate : boolean;`
	`attribute dont_replicate of my_reg : signal is true;`

Note: For compatibility with third-party synthesis tools, Quartus II Integrated Synthesis also supports the attribute `syn_replicate`. To disable replication, set `syn_replicate` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when you set the attribute to `1` or `true`.

Maximum Fan-Out

This **Maximum Fan-Out** attribute and logic option direct the Compiler to control the number of destinations that a node feeds. The Compiler duplicates a node and splits its fan-out until the individual fan-out of each copy falls below the maximum fan-out restriction. You can apply this option to a register or a logic cell buffer, or to a design entity that contains these elements. You can use this option to reduce the load of critical signals, which can improve performance. You can use the option to instruct the Compiler to duplicate a register that feeds nodes in different locations on the target device. Duplicating the register can enable the Fitter to place these new registers closer to their destination logic to minimize routing delay.

To turn off the option for a given node if you set the option at a higher level of the design hierarchy, in the **Netlist Optimizations** logic option, select **Never Allow**. If not disabled by the **Netlist Optimizations** option, the Compiler acknowledges the maximum fan-out constraint as long as the following conditions are met:

- The node is not part of a cascade, carry, or register cascade chain.
- The node does not feed itself.
- The node feeds other logic cells, DSP blocks, RAM blocks, and pins through data, address, clock enable, and other ports, but not through any asynchronous control ports (such as asynchronous clear).

The Compiler does not create duplicate nodes in these cases, because there is no clear way to duplicate the node, or to avoid the small differences in timing which could produce functional differences in the implementation (in the third condition above in which asynchronous control signals are involved). If you cannot apply the constraint because you do not meet one of these conditions, the Compiler issues a message to indicate that the Compiler ignores the maximum fan-out assignment. To instruct the Compiler not to check node destinations for possible problems such as the third condition, you can set the **Netlist Optimizations** logic option to **Always Allow** for a given node.

Note: If you have enabled any of the Quartus II netlist optimizations that affect registers, add the `preserve` attribute to any registers to which you have set a `maxfan` attribute. The `preserve` attribute ensures that the netlist optimization algorithms, such as register retiming, do not affect the registers.

You can set the **Maximum Fan-Out** logic option in the Quartus II software. This option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code, as shown in these examples. In these examples, the Compiler duplicates the `clk_gen` register, so its fan-out is not greater than 50.

Table 16-21: Setting the `maxfan` attribute in HDL Code

HDL	Code
Verilog HDL	`reg clk_gen /* synthesis syn_maxfan = 50 */;`
Verilog-2001	`(* maxfan = 50 *) reg clk_gen;`

Table 16-22: Setting the `syn_maxfan` attribute in HDL Code

The Quartus II software supports the `syn_maxfan` attribute for compatibility with other synthesis tools.

HDL	Code

VHDL	`signal clk_gen : stdlogic;`
	`attribute maxfan : signal ;`
	`attribute maxfan of clk_gen : signal is 50;`

Related Information

- **Netlist Optimizations and Physical Synthesis**
  For details about netlist optimizations
- **Maximum Fan-Out logic option**
  For more information about the Maximum Fan-Out logic option and the supported devices

Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable

The **Auto Clock Enable Replacement** logic option allows the software to find logic that feeds a register and move the logic to the register's clock enable input port. To solve fitting or performance issues with designs that have many clock enables, you can turn off this option for individual registers or design entities. Turning the option off prevents the software from using the register's clock enable port. The software implements the clock enable functionality using multiplexers in logic cells.

If the software does not move the specific logic to a clock enable input with the **Auto Clock Enable Replacement** logic option, you can instruct the software to use a direct clock enable signal. The attribute ensures that the signal drives the clock enable port, and the software does not optimize or combine the signal with other logic.

These tables show how to set this attribute to ensure that the attribute preserves the signal and uses the signal as a clock enable.

Table 16-23: Setting the `direct_enable` in HDL Code

HDL	Code
Verilog HDL	`wire my_enable /* synthesis direct_enable = 1 */ ;`

VHDL	`attribute direct_enable: boolean;`
	`attribute direct_enable of my_enable: signal is true;`

Table 16-24: Setting the `syn_direct_enable` in HDL Code

The Quartus II software supports the `syn_direct_enable` attribute name for compatibility with other synthesis tools.

HDL	Code
Verilog-2001 and SystemVerilog	`(* syn_direct_enable *) wire my_enable;`

Related Information

Auto Clock Enable Replacement logic option

For more information about the **Auto Clock Enable Replacement** logic option and the supported devices

Inferring Multiplier, DSP, and Memory Functions from HDL Code

The Quartus II Compiler automatically recognizes multipliers, multiply-accumulators, multiply-adders, or memory functions described in HDL code, and either converts the HDL code into respective IP core or maps them directly to device atoms or memory atoms. If the software converts the HDL code into an IP core, the software uses the Altera IP core code when you compile your design, even when you do not specifically instantiate the IP core. The software infers IP cores to take advantage of logic that you optimize for Altera devices. The area and performance of such logic can be better than the results from inferring generic logic from the same HDL code.

Additionally, you must use IP cores to access certain architecture-specific features, such as RAM, DSP blocks, and shift registers that provide improved performance compared with basic logic cells.

The Quartus II software provides options to control the inference of certain types of IP cores.

Related Information

- Recommended HDL Coding Styles on page 12-1
  For details about coding style recommendations when targeting IP cores in Altera devices

Multiply-Accumulators and Multiply-Adders

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. To disable inference, turn off this option for the entire project on the **Advanced Analysis & Synthesis** dialog box of the **Compiler Settings** page.

Related Information

Auto DSP Block Replacement logic option
For more information about the Auto DSP Block Replacement logic option and the supported devices

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option has three settings: **Off**, **Auto** and **Always**. **Auto** is the default setting in which Quartus II Integrated Synthesis decides which shift registers to replace or leave in registers. Placing shift registers in memory saves logic area, but can have a negative effect on f_{max}. Quartus II Integrated Synthesis uses the optimization technique setting, logic and RAM utilization of your design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are located in memory and which are located in registers. To disable inference, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**. You can also disable the option for a specific block with the Assignment Editor. Even if you set the logic option to **On** or **Auto**, the software might not infer small shift registers because small shift registers do not benefit from implementation in dedicated memory. However, you can use the **Allow Any Shift Register Size for Recognition** logic option to instruct synthesis to infer a shift register even when its size is too small.

You can use the **Allow Shift Register Merging across Hierarchies** option to prevent the Compiler from merging shift registers in different hierarchies into one larger shift register. The option has three settings: **On**, **Off**, and **Auto**. The **Auto** setting is the default setting, and the Compiler decides whether or not to merge shift registers across hierarchies. When you turn on this option, the Compiler allows all shift registers to merge across hierarchies, and when you turn off this option, the Compiler does not allow any shift registers to merge across hierarchies. You can set this option globally or on entities or individual nodes.

Note: The registers that the software maps to the RAM-based Shift Register IP core and places in RAM are not available in the Simulator because their node names do not exist after synthesis.

The Compiler turns off the **Auto Shift Register Replacement** logic option when you select a formal verification tool on the **EDA Tool Settings** page. If you do not select a formal verification tool, the Compiler issues a warning and the compilation report lists shift registers that the logic option might infer. To enable an IP core for the shift register in the formal verification flow, you can either instantiate a shift register explicitly with the IP catalog or make the shift register into a black box in a separate entity or module.

Related Information

Auto Shift Register Replacement logic option
For more information about the Auto Shift Register Replacement logic option and the supported devices

RAM-Based Shift Register (ALTSHIFT_TAPS) User Guide
For more information about the RAM-based Shift Register IP core

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. To disable the inference, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

Note: Although the software implements inferred shift registers in RAM blocks, you cannot turn off the **Auto RAM Replacement** option to disable shift register replacement. Use the **Auto Shift Register Replacement** option.

The software might not infer very small RAM or ROM blocks because you can implement very small memory blocks with the registers in the logic. However, you can use the **Allow Any RAM Size for Recognition** and **Allow Any ROM Size for Recognition** logic options to instruct synthesis to infer a memory block even when its size is too small.

Note: The software turns off the **Auto ROM Replacement** logic option when you select a formal verification tool in the **EDA Tool Settings** page. If you do not select a formal verification tool, the software issues a warning and a report panel provides a list of ROMs that the logic option might infer. To enable an IP core for the shift register in the formal verification flow, you can either instantiate a ROM explicitly using the IP Catalog or create a black box for the ROM in a separate entity or in a separate module.

Although formal verification tools do not support inferred RAM blocks, due to the importance of inferring RAM in many designs, the software turns on the **Auto RAM Replacement** logic option when you select a formal verification tool in the **EDA Tool Settings** page. The software automatically performs black box instance for any module or entity that contains an inferred RAM block. The software issues a warning and lists the black box created in the compilation report. This black box allows formal verification tools to proceed; however, the formal verification tool cannot verify the entire module or entire entity that contains the RAM. Altera recommends that you explicitly instantiate RAM blocks in separate modules or in separate entities so that the formal verification tool can verify as much logic as possible.

Related Information

- Shift Registers on page 16-45
- Auto RAM Replacement logic option
  For more information about the Auto RAM Replacement logic option and its supported devices
- Auto ROM Replacement logic option
  For more information about the Auto ROM Replacement logic option and its supported devices

Resource Aware RAM, ROM, and Shift-Register Inference

The Quartus II Integrated Synthesis considers resource usage when inferring RAM, ROM, and shift registers. During RAM, ROM, and shift register inferencing, synthesis looks at the number of memories available in the current device and does not infer more memory than is available to avoid a no-fit error.

Synthesis tries to select the memories that are not inferred in a way that aims at the smallest increase in logic and registers.

Resource aware RAM, ROM and shift register inference is controlled by the **Resource Aware Inference for Block RAM** option. To disable this option for the entire project, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

When you select the **Auto** setting, resource aware RAM, ROM, and shift register inference use the resource counts from the largest device.

For designs with multiple partitions, Quartus II Integrated Synthesis considers one partition at a time. Therefore, for each partition, it assumes that all RAM blocks are available to that partition. If this causes a no-fit error, you can limit the number of RAM blocks available per partition with the **Maximum Number of M512 Memory Blocks**, **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks**, **Maximum Number of M-RAM/M144K Memory Blocks and Maximum Number of LABs** settings in the Assignment Editor. The balancer also uses these options.

Auto RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** logic option allows Quartus II Integrated Synthesis to convert small RAM blocks to logic cells if the logic cell implementation gives better quality of results. The software converts only single-port or simple-dual port RAMs with no initialization files to logic cells. You can set this option globally or apply it to individual RAM nodes. You can enable this option by turning on the appropriate option for the entire project in the **Advanced Analysis & Synthesis Settings** dialog box.

For Arria GX and Stratix family of devices, the software uses the following rules to determine the placement of a RAM, either in logic cells or a dedicated RAM block:

- If the number of words is less than 16, use a RAM block if the total number of bits is greater than or equal to 64.
- If the number of words is greater than or equal to 16, use a RAM block if the total number of bits is greater than or equal to 32.
- Otherwise, implement the RAM in logic cells.

For the Cyclone family of devices, the software uses the following rules:

- If the number of words is greater than or equal to 64, use a RAM block.
- If the number of words is greater than or equal to 16 and less than 64, use a RAM block if the total number of bits is greater than or equal to 128.
- Otherwise, implement the RAM in logic cells.

Related Information

Auto RAM to Logic Cell Conversion logic option

For more information about the Auto RAM to Logic Cell Conversion logic options and the supported devices

RAM Style and ROM Style—for Inferred Memory

These attributes specify the implementation for an inferred RAM or ROM block. You can specify the type of TriMatrix embedded memory block, or specify the use of standard logic cells (LEs or ALMs). The Quartus II software supports the attributes only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The `M512`, `M4K`, `M-RAM`, `MLAB`, `M9K`, `M144K`, `M20K`, and `M10K` values (as applicable for the target device family) indicate the type of memory block to use for the inferred RAM or ROM. If you set the attribute to a block type that does not exist in the target device family, the software generates a warning and ignores the assignment. The `logic` value indicates that the Quartus II software implements the RAM or ROM in regular logic rather than dedicated memory blocks. You can set the attribute on a module or entity, in which case it specifies the default implementation style for all inferred memory blocks in the immediate hierarchy. You can also set the attribute on a specific signal (VHDL) or variable (Verilog HDL) declaration, in which case it specifies the preferred implementation style for that specific memory, overriding the default implementation style.

Note: If you specify a `logic` value, the memory appears as a RAM or ROM block in the RTL Viewer, but Integrated Synthesis converts the memory to regular logic during synthesis.

In addition to `ramstyle` and `romstyle`, the Quartus II software supports the `syn_ramstyle` attribute name for compatibility with other synthesis tools.

These tables specify that you must implement all memory in the module or the `my_memory_blocks` entity with a specific type of block.

Table 16-25: Applying a `romstyle` Attribute to a Module Declaration

HDL	Code
Verilog-1995	`module my_memory_blocks (...) /* synthesis romstyle = "M4K" */ ;`

Table 16-26: Applying a ramstyle Attribute to a Module Declaration

HDL	Code
Verilog-2001 and SystemVerilog	`(* ramstyle = "M512" *) module my_memory_blocks (...);`

Table 16-27: Applying a romstyle Attribute to an Architecture

HDL	Code

VHDL	`architecture rtl of my_ my_memory_blocks is`
	`attribute romstyle : string;`
	`attribute romstyle of rtl : architecture is "M-RAM";`
	`begin`

These tables specify that you must implement the inferred `my_ram` or `my_rom` memory with regular logic instead of a TriMatrix memory block.

Table 16-28: Applying a `syn_ramstyle` Attribute to a Variable Declaration

HDL	Code
Verilog-1995	`reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */ ;`

Table 16-29: Applying a `romstyle` Attribute to a Variable Declaration

HDL	Code
Verilog-2001 and SystemVerilog	`(* romstyle = "logic" *) reg [0:7] my_rom[0:63];`

Table 16-30: Applying a `ramstyle` Attribute to a Signal Declaration

HDL	Code

VHDL	`type memory_t is array (0 to 63) of std_logic_vector (0 to 7) ;`
	`signal my_ram : memory_t;`
	`attribute ramstyle : string;`
	`attribute ramstyle of my_ram : signal is "logic";`

You can control the depth of an inferred memory block and optimize its usage with the `max_depth` attribute. You can also optimize the usage of the memory block with this attribute.

These tables specify the depth of the inferred memory `mem` using the `max_depth` synthesis attribute.

Table 16-31: Applying a max_depth Attribute to a Variable Declaration

HDL	Code
Verilog-1995	`reg [7:0] mem [127:0] /* synthesis max_depth = 2048 */`

Table 16-32: Applying a max_depth Attribute to a Variable Declaration

HDL	Code
Verilog-2001 and SystemVerilog	`(* max_depth = 2048*) reg [7:0] mem [127:0];`

Table 16-33: Applying a max_depth Attribute to a Variable Declaration

HDL	Code

VHDL	`type ram_block is array (0 to 31) of std_logic_vector (2 downto 0);`
	`signal mem : ram_block;`
	`attribute max_depth : natural;`
	`attribute max_depth OF mem : signal is 2048;`

The syntax for setting these attributes in HDL is the same as the syntax for other synthesis attributes, as shown in Synthesis Attributes on page 16-22.

Related Information

Synthesis Attributes on page 16-22

RAM Style Attribute—For Shift Registers Inference

The RAM style attribute for shift register allows you to use the RAM style attribute for shift registers, just as you use them for RAM or ROMs. The Quartus II Synthesis uses the RAM style attribute during shift register inference. If synthesis infers the shift register to RAM, it will be sent to the requested RAM block type. Shift registers are merged only if the RAM style attributes are compatible. If the RAM style is set to logic, a shift register does not get inferred to RAM.

Table 16-34: Setting the RAM Style Attribute for Shift Registers

HDL	Code
Verilog	`(* ramstyle = "mlab" *)reg [N-1:0] sr;`
VHDL	`attribute ramstyle : string;attribute ramstyle of sr : signal is "M20K";`

Related Information

Inferring Shift Registers in HDL Code on page 12-30

Disabling Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute

Use the `no_rw_check` value for the `ramstyle` attribute, or disable the **Add Pass-Through Logic to Inferred RAMs** logic option assignment to indicate that your design does not depend on the behavior of the inferred RAM, when there are reads and writes to the same address in the same clock cycle. If you specify the attribute or disbale the logic option, the Quartus II software chooses a read-during-write behavior instead of the read-during-write behavior of your HDL source code.

You disable or edit the attributes of this option by modifying the `add_pass_through_logic_to_inferred_rams option` in the Quartus II Settings File (**.qsf**). There is no corresponding GUI setting for this option.

Sometimes, you must map an inferred RAM into regular logic cells because the inferred RAM has a read-during-write behavior that the TriMatrix memory blocks in your target device do not support. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source to increase the area of your design and potentially reduce its performance. In some of these cases, you can use the attribute to specify that the software can implement the RAM directly in a TriMatrix memory block without using logic. You can also use the attribute to prevent a warning message for dual-clock RAMs in the case that the inferred behavior in the device does not exactly match the read-during-write conditions described in the HDL code.

To set the **Add Pass-Through Logic to Inferred RAMs** logic option with the Quartus II software, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

These examples use two addresses and normally require extra logic after the RAM to ensure that the read-during-write conditions in the device match the HDL code. If your design does not require a defined read-during-write condition, the extra logic is not necessary. With the `no_rw_check` attribute, Quartus II Integrated Synthesis does not generate the extra logic.

Table 16-35: Inferred RAM Using `no_rw_check` Attribute

HDL	Code

Verilog HDL	`module ram_infer (q, wa, ra, d, we, clk);`
	`    output [7:0] q;`
	`    input [7:0] d;`
	`    input [6:0] wa;`
	`    input [6:0] ra;`
	`    input we, clk;`
	`    reg [6:0] read_add;`
	`    (* ramstyle = "no_rw_check" *) reg [7:0] mem [127:0];`
	`    always @ (posedge clk) begin`
	`        if (we)`
	`            mem[wa] <= d;`
	`        read_add <= ra;`
	`    end`
	`    assign q = mem[read_add];`
	`endmodule`

VHDL	`LIBRARY ieee;`
	`USE ieee.std_logic_1164.ALL;`
	`ENTITY ram IS`
	`    PORT (`
	`        clock: IN STD_LOGIC;`
	`        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);`
	`        write_address: IN INTEGER RANGE 0 to 31;`
	`        read_address: IN INTEGER RANGE 0 to 31;`
	`        we: IN STD_LOGIC;`
	`        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)    );`
	`END ram;`
	`ARCHITECTURE rtl OF ram IS`
	`    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);`
	`    SIGNAL ram_block: MEM;`
	`    ATTRIBUTE ramstyle : string;`
	`    ATTRIBUTE ramstyle of ram_block : signal is "no_rw_check";`
	`    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;`
	`BEGIN`
	`    PROCESS (clock)`
	`    BEGIN`
	`        IF (clock'event AND clock = '1') THEN`
	`            IF (we = '1') THEN`
	`                ram_block(write_address) <= data;`
	`            END IF;`
	`            read_address_reg <= read_address;`
	`        END IF;`
	`    END PROCESS;`
	`    q <= ram_block(read_address_reg);`
	`END rtl;`

You can use a ramstyle attribute with the MLAB value, so that the Quartus II software can infer a small RAM block and place it in an MLAB.

Note: You can use this attribute in cases in which some asynchronous RAM blocks might be coded with read-during-write behavior that does not match the Stratix IV and Stratix V architectures. Thus, the device behavior would not exactly match the behavior that the code describes. If the difference in behavior is acceptable in your design, use the `ramstyle` attribute with the `no_rw_check` value to specify that the software should not check the read-during-write behavior when inferring the RAM. When you set this attribute, Quartus II Integrated Synthesis allows the behavior of the output to differ when the asynchronous read occurs on an address that had a write on the most recent clock edge. That is, the functional HDL simulation results do not match the hardware behavior if you write to an address that is being read. To include these attributes, set the value of the ramstyle attribute to `MLAB, no_rw_check`.

These examples show the method of setting two values to the `ramstyle` attribute with a small asynchronous RAM block, with the ramstyle synthesis attribute set, so that the software can implement the memory in the MLAB memory block and so that the read-during-write behavior is not important. Without the attribute, this design requires 512 registers and 240 ALUTs. With the attribute, the design requires eight memory ALUTs and only 15 registers.

Table 16-36: Inferred RAM Using `no_rw_check` and MLAB Attributes

HDL	Code

Verilog HDL	`module async_ram (`
	`    input   [5:0] addr,`
	`    input   [7:0] data_in,`
	`    input          clk,`
	`    input          write,`
	`    output  [7:0] data_out );`
	`   (* ramstyle = "MLAB, no_rw_check" *) reg [7:0] mem[0:63];`
	`   assign  data_out = mem[addr];`
	`   always @ (posedge clk)`
	`   begin`
	`      if (write)`
	`         mem[addr] = data_in;`
	`   end`
	`endmodule`

VHDL	`LIBRARY ieee;`
	`USE ieee.std_logic_1164.ALL;`
	`ENTITY ram IS`
	`    PORT (`
	`        clock: IN STD_LOGIC;`
	`        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);`
	`        write_address: IN INTEGER RANGE 0 to 31;`
	`        read_address: IN INTEGER RANGE 0 to 31;`
	`        we: IN STD_LOGIC;`
	`        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));`
	`END ram;`
	`ARCHITECTURE rtl OF ram IS`
	`    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);`
	`    SIGNAL ram_block: MEM;`
	`    ATTRIBUTE ramstyle : string;`
	`    ATTRIBUTE ramstyle of ram_block : signal is "MLAB , no_rw_`
	`check";`
	`    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;`
	`BEGIN`
	`    PROCESS (clock)`
	`    BEGIN`
	`        IF (clock'event AND clock = '1') THEN`
	`            IF (we = '1') THEN`
	`                ram_block(write_address) <= data;`
	`            END IF;`
	`            read_address_reg <= read_address;`
	`        END IF;`
	`    END PROCESS;`
	`    q <= ram_block(read_address_reg);`
	`END rtl;`

Related Information

Recommended HDL Coding Styles on page 12-1
For more information about recommended styles for inferring RAM and some of the issues involved with different read-during-write conditions

Add Pass-Through Logic to Inferred RAMs logic option
For more information about the Add Pass-Through Logic to Inferred RAMs logic option and the supported devices

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory with a **.mif**. The attribute takes a string value containing the name of the RAM initialization file.

The `ram_init_file` attribute is supported for ROM too.

Table 16-37: Applying a ram_init_file Attribute

HDL	Code
Verilog-1995	`reg [7:0] mem[0:255] /* synthesis ram_init_file = " my_init_file.mif" */;`
Verilog-2001	`(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];`

VHDL[12]	`type mem_t is array(0 to 255) of unsigned(7 downto 0);`
	`signal ram : mem_t;`
	`attribute ram_init_file : string;`
	`attribute ram_init_file of ram :`
	`signal is "my_init_file.mif";`

Related Information

- **Recommended HDL Coding Styles** on page 12-1
  For more information about Inferring ROM Functions from HDL Code

Multiplier Style—for Inferred Multipliers

The `multstyle` attribute specifies the implementation style for multiplication operations (`*`) in your HDL source code. You can use this attribute to specify whether you prefer the Compiler to implement a multiplication operation in general logic or dedicated hardware, if available in the target device.

The `multstyle` attribute takes a string value of `"logic"` or `"dsp"`, indicating a preferred implementation in logic or in dedicated hardware, respectively. In Verilog HDL, apply the attribute to a module declaration, a variable declaration, or a specific binary expression that contains the `*` operator. In VHDL, apply the synthesis attribute to a signal, variable, entity, or architecture.

[12] You can also initialize the contents of an inferred memory by specifying a default value for the corresponding signal. In Verilog HDL, you can use an initial block to specify the memory contents. Quartus II Integrated Synthesis automatically converts the default value into a **.mif** for the inferred RAM.

Note: Specifying a `multstyle` of `"dsp"` does not guarantee that the Quartus II software can implement a multiplication in dedicated DSP hardware. The final implementation depends on several conditions, including the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

In addition to `multstyle`, the Quartus II software supports the `syn_multstyle` attribute name for compatibility with other synthesis tools.

When applied to a Verilog HDL module declaration, the attribute specifies the default implementation style for all instances of the `*` operator in the module. For example, in the following code examples, the `multstyle` attribute directs the Quartus II software to implement all multiplications inside module `my_module` in the dedicated multiplication hardware.

Table 16-38: Applying a `multstyle` Attribute to a Module Declaration

HDL	Code
Verilog-1995	`module my_module (...) /* synthesis multstyle = "dsp" */;`
Verilog-2001	`(* multstyle = "dsp" *) module my_module(...);`

When applied to a Verilog HDL variable declaration, the attribute specifies the implementation style for a multiplication operator, which has a result directly assigned to the variable. The attribute overrides the `multstyle` attribute with the enclosing module, if present.

In these examples, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in logic rather than the dedicated hardware.

Table 16-39: Applying a `multstyle` Attribute to a Variable Declaration

HDL	Code

Verilog-2001	`wire [8:0] a, b;`
	`(* multstyle = "logic" *) wire [17:0] result;`
	`assign result = a * b;  //Multiplication must be`
	`                        //directly assigned to result`

Verilog-1995	`wire [8:0] a, b;`
	`wire [17:0] result /* synthesis multstyle = "logic" */;`
	`assign result = a * b;  //Multiplication must be`
	
	`                        //directly assigned to result`

When applied directly to a binary expression that contains the `*` operator, the attribute specifies the implementation style for that specific operator alone and overrides any `multstyle` attribute with the target variable or enclosing module.

In this example, the `multstyle` attribute indicates that you must implement `a * b` in the dedicated hardware.

Table 16-40: Applying a `multstyle` Attribute to a Binary Expression

HDL	Code

Verilog-2001	`wire [8:0] a, b;`
	`wire [17:0] result;`
	`assign result = a * (* multstyle = "dsp" *) b;`

Note: You cannot use Verilog-1995 attribute syntax to apply the `multstyle` attribute to a binary expression.

When applied to a VHDL entity or architecture, the attribute specifies the default implementation style for all instances of the `*` operator in the entity or architecture.

In this example, the `multstyle` attribute directs the Quartus II software to use dedicated hardware, if possible, for all multiplications inside architecture `rtl` of entity `my_entity`.

Table 16-41: Applying a `multstyle` Attribute to an Architecture

HDL	Code

VHDL	`architecture rtl of my_entity is`
	`    attribute multstyle : string;`
	`    attribute multstyle of rtl : architecture is "dsp";`
	`begin`

When applied to a VHDL signal or variable, the attribute specifies the implementation style for all instances of the `*` operator, which has a result directly assigned to the signal or variable. The attribute overrides the `multstyle` attribute with the enclosing entity or architecture, if present.

In this example, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in logic rather than the dedicated hardware.

Table 16-42: Applying a `multstyle` Attribute to a Signal or Variable

HDL	Code

VHDL	`signal a, b : unsigned(8 downto 0);`
	`signal result : unsigned(17 downto 0);`
	
	`attribute multstyle : string;`
	`attribute multstyle of result : signal is "logic";`
	`result <= a * b;`

Full Case Attribute

A Verilog HDL case statement is full when its case items cover all possible binary values of the case expression or when a default case statement is present. A `full_case` attribute attached to a case statement header that is not full forces synthesis to treat the unspecified states as a don't care value. VHDL case statements must be full, so the attribute does not apply to VHDL.

Using this attribute on a case statement that is not full allows you to avoid the latch inference problems.

Note: Latches have limited support in formal verification tools. Do not infer latches unintentionally, for example, through an incomplete case statement when using formal verification.

Formal verification tools support the `full_case` synthesis attribute (with limited support for attribute syntax, as described in **Synthesis Attributes** on page 16-22).

Using the `full_case` attribute might cause a simulation mismatch between the Verilog HDL functional and the post-Quartus II simulation because unknown case statement cases can still function as latches during functional simulation. For example, a simulation mismatch can occur with the code in **Table 16-43** when `sel` is `2'b11` because a functional HDL simulation output behaves as a latch and the Quartus II simulation output behaves as a don't care value.

Note: Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

Table 16-43: A full_case Attribute

The case statement in this example is not full because you do not specify some `sel` binary values. Because you use the `full_case` attribute, synthesis treats the output as "don't care" when the `sel` input is `2'b11`.

HDL	Code

Verilog HDL	
	```
	module full_case (a, sel, y);
	    input [3:0] a;
	    input [1:0] sel;
	    output y;
	    reg y;
	    always @ (a or sel)
	    case (sel)             // synthesis full_case
	        2'b00: y=a[0];
	        2'b01: y=a[1];
	        2'b10: y=a[2];
	   endcase
	endmodule
	```

Verilog-2001 syntax also accepts the statements in **Table 16-44** in the `case` header instead of the comment form as shown in **Table 16-43**.

Table 16-44: Syntax for the full_case Attribute

HDL	Syntax
Verilog-2001	`(* full_case *) case (sel)`

Related Information

Synthesis Attributes on page 16-22

Recommended Design Practices on page 11-1
For more information about avoiding latch inference problems

Parallel Case

The `parallel_case` attribute indicates that you must consider a Verilog HDL case statement as parallel; that is, you can match only one case item at a time. Case items in Verilog HDL case statements might overlap. To resolve multiple matching case items, the Verilog HDL language defines a priority among case items in which the case statement always executes the first case item that matches the case expression value. By default, the Quartus II software implements the extra logic necessary to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement header allows the Quartus II software to consider its case items as inherently parallel; that is, at most one case item matches the case expression value. Parallel case items simplify the generated logic.

In VHDL, the individual choices in a case statement might not overlap, so they are always parallel and this attribute does not apply.

Altera recommends that you use this attribute only when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic does not match the functional simulation behavior of the Verilog HDL.

Note: Altera recommends that you avoid using the `parallel_case` attribute, because you may mismatch the Verilog HDL functional and the post-Quartus II simulation.

If you specify SystemVerilog-2005 as the supported Verilog HDL version for your design, you can use the SystemVerilog keyword `unique` to achieve the same result as the `parallel_case` directive without causing simulation mismatches.

This example shows a `casez` statement with overlapping case items. In functional HDL simulation, the software prioritizes the three case items by the bits in `sel`. For example, `sel[2]` takes priority over `sel[1]`, which takes priority over `sel[0]`. However, the synthesized design can simulate differently because the `parallel_case` attribute eliminates this priority. If more than one bit of `sel` is high, more than one output (`a`, `b`, or `c`) is high as well, a situation that cannot occur in functional HDL simulation.

Table 16-45: A `parallel_case` Attribute

HDL	Code

Verilog HDL	
	```
	module parallel_case (sel, a, b, c);
	    input [2:0] sel;
	    output a, b, c;
	    reg a, b, c;
	    always @ (sel)
	    begin
	        {a, b, c} = 3'b0;
	        casez (sel)          // synthesis parallel_case
	            3'b1??: a = 1'b1;
	            3'b?1?: b = 1'b1;
	            3'b??1: c = 1'b1;
	        endcase
	    end
	endmodule
	```

Table 16-46: Verilog-2001 Syntax

Verilog-2001 syntax also accepts the statements as shown in the following table in the `case` (or `casez`) header instead of the comment form, as shown in **Table 16-45**.

HDL	Syntax
Verilog-2001	`(* parallel_case *) casez (sel)`

Translate Off and On / Synthesis Off and On

The `translate_off` and `translate_on` synthesis directives indicate whether the Quartus II software or a third-party synthesis tool should compile a portion of HDL code that is not relevant for synthesis. The `translate_off` directive marks the beginning of code that the synthesis tool should ignore; the `translate_on` directive indicates that synthesis should resume. You can also use the `synthesis_on` and `synthesis_off` directives as a synonym for translate on and off.

You can use these directives to indicate a portion of code for simulation only. The synthesis tool reads synthesis-specific directives and processes them during synthesis; however, third-party simulation tools read the directives as comments and ignore them.

These examples show these directives.

Table 16-47: Translate Off and On

HDL	Code

Verilog HDL	`// synthesis translate_off`
	`parameter tpd = 2;    // Delay for simulation`
	`#tpd;`
	`// synthesis translate_on`

VHDL	`-- synthesis translate_off`
	`use std.textio.all;`
	`-- synthesis translate_on`

VHDL 2008	`/* synthesis translate_off */`
	`use std.textio.all;`
	`/* synthesis translate_on */`

If you want to ignore only a portion of code in Quartus II Integrated Synthesis, you can use the Altera-specific attribute keyword `altera`. For example, use the `// altera translate_off` and `// altera translate_on` directives to direct Quartus II Integrated Synthesis to ignore a portion of code that you intend only for other synthesis tools.

Ignore translate_off and synthesis_off Directives

The **Ignore translate_off and synthesis_off Directives** logic option directs Quartus II Integrated Synthesis to ignore the `translate_off` and `synthesis_off` directives. Turning on this logic option allows you to compile code that you want the third-party synthesis tools to ignore; for example, IP core declarations that the other tools treat as black boxes but the Quartus II software can compile. To set the

Ignore translate_off and synthesis_off Directives logic option, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

Related Information

Ignore translate_off and synthesis_off Directives logic option
For more information about the **Ignore translate_off and synthesis_off Directives** logic option and the supported devices

Read Comments as HDL

The `read_comments_as_HDL` synthesis directive indicates that the Quartus II software should compile a portion of HDL code that you commented out. This directive allows you to comment out portions of HDL source code that are not relevant for simulation, while instructing the Quartus II software to read and synthesize that same source code. Setting the `read_comments_as_HDL` directive to on indicates the beginning of commented code that the synthesis tool should read; setting the `read_comments_as_HDL` directive to `off` indicates the end of the code.

Note: You can use this directive with `translate_off` and `translate_on` to create one HDL source file that includes an IP core instantiation for synthesis and a behavioral description for simulation.

Formal verification tools do not support the `read_comments_as_HDL` directive because the tools do not recognize the directive.

In these examples, the Compiler synthesizes the commented code enclosed by `read_comments_as_HDL` because the directive is visible to the Quartus II Compiler. VHDL 2008 allows block comments, which comments are also supported for synthesis directives.

Note: Because synthesis directives are case sensitive in Verilog HDL, you must match the case of the directive, as shown in the following examples.

Table 16-48: Read Comments as HDL

HDL	Code

Verilog HDL	`// synthesis read_comments_as_HDL on`
	`// my_rom lpm_rom     (.address (address),`
	`//                       .data    (data));`
	`// synthesis read_comments_as_HDL off`

VHDL	`-- synthesis read_comments_as_HDL on`
	`-- my_rom : entity lpm_rom`
	`--   port map (`
	`--    address => address,`
	`--    data    => data,      );`
	`-- synthesis read_comments_as_HDL off`

VHDL 2008	`/* synthesis read_comments_as_HDL on */`
	`/* my_rom : entity lpm_rom`
	`    port map (`
	`    address => address,`
	`    data => data, ); */`
	`    synthesis read_comments_as_HDL off */`

Use I/O Flipflops

The useioff attribute directs the Quartus II software to implement input, output, and output enable flipflops (or registers) in I/O cells that have fast, direct connections to an I/O pin, when possible. To improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times, you can apply the useioff synthesis attribute. The **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options support this synthesis attribute. You can also set this synthesis attribute in the Assignment Editor.

The useioff synthesis attribute takes a boolean value. You can apply the value only to the port declarations of a top-level Verilog HDL module or VHDL entity (it is ignored if applied elsewhere). Setting the value to 1 (Verilog HDL) or TRUE (VHDL) instructs the Quartus II software to pack registers into I/O cells. Setting the value to 0 (Verilog HDL) or FALSE (VHDL) prevents register packing into I/O cells.

In Table 16-49 and Table 16-50, the useioff synthesis attribute directs the Quartus II software to implement the a_reg, b_reg, and o_reg registers in the I/O cells corresponding to the a, b, and o ports, respectively.

Table 16-49: Verilog HDL Code: The useioff Attribute

HDL	Code
Verilog HDL	see code below

```
module top_level(clk, a, b, o);
       input clk;
       input [1:0] a, b /* synthesis useioff = 1 */;
       output [2:0] o /* synthesis useioff = 1 */;
       reg [1:0] a_reg, b_reg;
       reg [2:0] o_reg;
       always @ (posedge clk)
       begin
           a_reg <= a;
           b_reg <= b;
           o_reg <= a_reg + b_reg;
       end
       assign o = o_reg;
endmodule
```

Table 16-50 and Table 16-51 show that the Verilog-2001 syntax also accepts the type of statements instead of the comment form in Table 16-49.

Table 16-50: Verilog-2001 Code: the useioff Attribute

HDL	Code
Verilog-2001	see code below

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Table 16-51: VHDL Code: the useioff Attribute

HDL	Code

VHDL	
	```
	library ieee;
	use ieee.std_logic_1164.all;
	use ieee.numeric_std.all;
	entity useioff_example is
	   port (
	      clk  : in  std_logic;
	      a, b : in  unsigned(1 downto 0);
	      o    : out unsigned(1 downto 0));
	   attribute useioff : boolean;
	   attribute useioff of a : signal is true;
	   attribute useioff of b : signal is true;
	   attribute useioff of o : signal is true;
	end useioff_example;
	architecture rtl of useioff_example is
	   signal o_reg, a_reg, b_reg : unsigned(1 downto 0);
	begin
	   process(clk)
	   begin
	      if (clk = '1' AND clk'event) then
	          a_reg <= a;
	          b_reg <= b;
	          o_reg <= a_reg + b_reg;
	      end if;
	   end process;
	o <= o_reg;
	end rtl;
	```

Specifying Pin Locations with chip_pin

The `chip_pin` attribute allows you to assign pin locations in your HDL source. You can use the attribute only on the ports of the top-level entity or module in your design. You can assign pins only to single-bit or one-dimensional bus ports in your design.

For single-bit ports, the value of the `chip_pin` attribute is the name of the pin on the target device, as specified by the pin table of the device.

Note: In addition to the `chip_pin` attribute, the Quartus II software supports the `altera_chip_pin_lc` attribute name for compatibility with other synthesis tools. When using this attribute in other synthesis tools, some older device families require an “@” symbol in front of each pin assignment. In the Quartus II software, the “@” is optional.

Table 16-52: Applying Chip Pin to a Single Pin

These examples in this table show different ways of assigning `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

HDL	Code

Verilog-1995	
	```
	input my_pin1 /* synthesis chip_pin = "C1" */;
	input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
	```

Verilog-2001	
	```
	(* chip_pin = "C1" *) input my_pin1;
	(* altera_chip_pin_lc = "@4" *) input my_pin2;
	```

HDL	Code

VHDL	`entity my_entity is`
	`port(my_pin1: in std_logic; my_pin2: in std_logic;…);`
	`end my_entity;`
	`attribute chip_pin : string;`
	`attribute altera_chip_pin_lc : string;`
	`attribute chip_pin of my_pin1 : signal is "C1";`
	`attribute altera_chip_pin_lc of my_pin2 : signal is "@4";`

For bus I/O ports, the value of the chip pin attribute is a comma-delimited list of pin assignments. The order in which you declare the range of the port determines the mapping of assignments to individual bits in the port. To leave a bit unassigned, leave its corresponding pin assignment blank.

Table 16-53: Applying Chip Pin to a Bus of Pins

The example in this table assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

HDL	Code
Verilog-1995	`input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;`

Table 16-54: Applying Chip Pin to Part of a Bus

The example in this table reverses the order of the signals in the bus, assigning `my_pin[0]` to `Pin_4` and `my_pin[2]` to `Pin_6` but leaves `my_pin[1]` unassigned.

HDL	Code
Verilog-1995	`input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;`

Table 16-55: Applying Chip Pin to Part of a Bus of Pins

The example in this table assigns `my_pin[2]` to Pin `4` and `my_pin[0]` to Pin `6`, but leaves `my_pin[1]` unassigned.

HDL	Code

VHDL	`entity my_entity is`
	`port(my_pin: in std_logic_vector(2 downto 0);…);`
	`end my_entity;`
	`attribute chip_pin of my_pin: signal is "4, , 6";`

Table 16-56: VHDL and Verilog-2001 Examples: Assigning Pin Location and I/O Standard

HDL	Code

VHDL	`attribute altera_chip_pin_lc: string;`
	`attribute  altera_attribute: string;`
	`attribute altera_chip_pin_lc of clk: signal is "B13";`
	`attribute altera_attribute of clk:signal is "-name IO_STANDARD ""3.3-`
	`V LVCMOS""";`

HDL	Code

Verilog-2001	`(* altera_attribute = "-name IO_STANDARD \"3.3-V LVCMOS\"" *)(* chip_pin = "L5" *)input clk;`
	`(* altera_attribute = "-name IO_STANDARD LVDS" *)(* chip_pin = "L4" *)input sel;`
	`output [3:0] data_o, input [3:0] data_i);`

Using altera_attribute to Set Quartus II Logic Options

The `altera_attribute` attribute allows you to apply Quartus II logic options and assignments to an object in your HDL source code. You can set this attribute on an entity, architecture, instance, register, RAM block, or I/O pin. You cannot set it on an arbitrary combinational node such as a net. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL synthesis attribute. You can also use this attribute to pass entity-level settings and assignments to phases of the Compiler flow that follow Analysis & Synthesis, such as Fitting.

Assignments or settings made through the Quartus II software, the **.qsf**, or the Tcl interface take precedence over assignments or settings made with the `altera_attribute` synthesis attribute in your HDL code.

The attribute value is a single string containing a list of **.qsf** variable assignments separated by semicolons:

```
-name <variable_1> <value_1>;-name <variable_2> <value_2>[;…]
```

If the Quartus II option or assignment includes a target, source, and section tag, you must use the syntax in this example for each **.qsf** variable assignment:

```
-name <variable> <value>
-from <source> -to <target> -section_id <section>
```

This example shows the syntax for the full attribute value, including the optional target, source, and section tags for two different **.qsf** assignments:

```
" -name <variable_1> <value_1> [-from <source_1>] [-to <target_1>] [-section_id \
<section_1>]; -name <variable_2> <value_2> [-from <source_2>] [-to <target_2>] \
[-section_id <section_2>] "
```

Table 16-57: Example Usage

If the assigned value of a variable is a string of text, you must use escaped quotes around the value in Verilog HDL or double-quotes in VHDL:

HDL	Code
Assigned Value of a Variable in Verilog HDL (With Nonexistent Variable and Value Terms)	`"VARIABLE_NAME \"STRING_VALUE\""`
Assigned Value of a Variable in VHDL (With Nonexistent Variable and Value Terms)	`"VARIABLE_NAME ""STRING_VALUE"""`

To find the **.qsf** variable name or value corresponding to a specific Quartus II option or assignment, you can set the option setting or assignment in the Quartus II software, and then make the changes in the **.qsf**.

Applying altera_attribute to an Instance

These examples use `altera_attribute` to set the power-up level of an inferred register.

Table 16-58: Applying altera_attribute to an Instance

These examples use `altera_attribute` to set the power-up level of an inferred register.

HDL	Code
Verilog-1995	`reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH" */;`
Verilog-2001	`(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;`

VHDL	`signal my_reg : std_logic;`
	`attribute altera_attribute : string;`
	`attribute altera_attribute of my_reg: signal is "-name POWER_UP_LEVEL HIGH";`

Note: For inferred instances, you cannot apply the attribute to the instance directly. Therefore, you must apply the attribute to one of the output nets of the instance. The Quartus II software automatically moves the attribute to the inferred instance.

Applying altera_attribute to an Entity

These examples use the `altera_attribute` to disable the **Auto Shift Register Replacement** synthesis option for an entity. To apply the Altera Attribute to a VHDL entity, you must set the attribute on its architecture rather than on the entity itself.

Table 16-59: Applying altera_attribute to an Entity

HDL	Code
Verilog-1995	`module my_entity(…) /* synthesis altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;`
Verilog-2001	`(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *) module my_entity(…) ;`

VHDL	`entity my_entity is`
	`-- Declare generics and ports`
	`end my_entity;`
	`architecture rtl of my_entity is`
	`attribute altera_attribute : string;`
	`-- Attribute set on architecture, not entity`
	`attribute altera_attribute of rtl: architecture is "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF";`
	`begin`
	`-- The architecture body`
	`end rtl;`

Applying altera_attribute with the -to Option

You can also use `altera_attribute` for more complex assignments that have more than one instance. In Table 16-60, the `altera_attribute` cuts all timing paths from `reg1` to `reg2`, equivalent to this Tcl or **.qsf** command, as shown in the example below:

```
set_instance_assignment -name CUT ON -from reg1 -to reg2
```

Table 16-60: Applying altera_attribute with the -to Option

HDL	Code

Verilog-1995	`reg reg2;`
	`reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;`

Verilog-2001 and SystemVerilog	`reg reg2;`
	`(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;`

VHDL	`signal reg1, reg2 : std_logic;`
	`attribute altera_attribute: string;`
	`attribute altera_attribute of reg1 : signal is "-name CUT ON -to reg2";`

You can specify either the `-to` option or the `-from` option in a single `altera_attribute`; Integrated Synthesis automatically sets the remaining option to the target of the `altera_attribute`. You can also specify wildcards for either option. For example, if you specify "`*`" for the `-to` option instead of `reg2` in these examples, the Quartus II software cuts all timing paths from `reg1` to every other register in this design entity.

You can use the `altera_attribute` only for entity-level settings, and the assignments (including wildcards) apply only to the current entity.

Related Information

Synthesis Attributes on page 16-22

Quartus II Settings File Manual
Lists all variable names

Analyzing Synthesis Results

After performing synthesis, you can check your synthesis results in the **Analysis & Synthesis** section of the Compilation Report and the Project Navigator.

Analysis & Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation. After Analysis & Synthesis, the Summary section of the Compilation Report provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred. The **Analysis & Synthesis** section lists synthesis-specific information.

Analysis & Synthesis includes various report sections, including a list of the source files read for the project, the resource utilization by entity after synthesis, and information about state machines, latches, optimization results, and parameter settings.

Related Information

Analysis Synthesis Summary Reports
For more information about each report section

Project Navigator

The **Hierarchy** tab of the Project Navigator provides a view of the project hierarchy and a summary of resource and device information about the current project. After Analysis & Synthesis, before the Fitter begins, the Project Navigator provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred.

If an entity in the Hierarchy tab contains parameter settings, a tooltip displays the settings when you hold the pointer over the entity.

Upgrade IP Components Dialog Box

In the Quartus II software version 12.1 SP1 and later, the **Upgrade IP Components** dialog box allows you to upgrade all outdated IP in your project after you move to a newer version of the Quartus II software.

Related Information

Upgrade IP Components dialog box
For more information about the Upgrade IP Components dialog box

Analyzing and Controlling Synthesis Messages

You can analyze the generated messages during synthesis and control which messages appear during compilation.

Quartus II Messages

The messages that appear during Analysis & Synthesis describe many of the optimizations during the synthesis stage, and provide information about how the software interprets your design. Altera recommends checking the messages to analyze **Critical Warnings** and **Warnings**, because these messages can relate to important design problems. Read the **Info** messages to get more information about how the software processes your design.

The software groups the messages by following types: **Info**, **Warning**, **Critical Warning**, and **Error**.

You can specify the type of Analysis & Synthesis messages that you want to view by selecting the **Analysis & Synthesis Message Level** option. To specify the display level, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**

Related Information

- **About the Messages Window**
  For more information about the Messages window and message suppression
- **About Message Suppression**
  For more information about the Messages window and message suppression

- **Managing Quartus II Projects** on page 1-1
  For more information about the Messages

VHDL and Verilog HDL Messages

The Quartus II software issues a variety of messages when it is analyzing and elaborating the Verilog HDL and VHDL files in your design. These HDL messages are a subset of all Quartus II messages that help you identify potential problems early in the design process.

HDL messages fall into the following categories:

- **Info message**—lists a property of your design.
- **Warning message**—indicates a potential problem in your design. Potential problems come from a variety of sources, including typos, inappropriate design practices, or the functional limitations of your target device. Though HDL warning messages do not always identify actual problems, Altera recommends investigating code that generates an HDL warning. Otherwise, the synthesized behavior of your design might not match your original intent or its simulated behavior.
- **Error message**—indicates an actual problem with your design. Your HDL code can be invalid due to a syntax or semantic error, or it might not be synthesizable as written.

In this example, the sensitivity list contains multiple copies of the variable `i`. While the Verilog HDL language does not prohibit duplicate entries in a sensitivity list, it is clear that this design has a typing error: Variable `j` should be listed on the sensitivity list to avoid a possible simulation or synthesis mismatch.

```
//dup.v
module dup(input i, input j, output reg o);
always @ (i or i)
    o = i & j;
endmodule
```

When processing the HDL code, the Quartus II software generates the following warning message.

```
Warning: (10276) Verilog HDL sensitivity list warning at dup.v(2): sensitivity list
contains multiple entries for "i".
```

In Verilog HDL, variable names are case sensitive, so the variables `my_reg` and `MY_REG` below are two different variables. However, declaring variables that have names in different cases is confusing, especially if you use VHDL, in which variables are not case sensitive.

```
// namecase.v
module namecase (input i, output o);
    reg my_reg;
    reg MY_REG;
    assign o = i;
endmodule
```

When processing the HDL code, the Quartus II software generates the following informational message:

```
Info: (10281) Verilog HDL information at namecase.v(3): variable name "MY_REG" and
variable name "my_reg" should not differ only in case.
```

In addition, the Quartus II software generates additional HDL info messages to inform you that this small design does not use neither `my_reg` nor `MY_REG`:

```
Info: (10035) Verilog HDL or VHDL information at namecase.v(3): object "my_reg"
declared but not used
Info: (10035) Verilog HDL or VHDL information at namecase.v(4): object "MY_REG"
declared but not used
```

The Quartus II software allows you to control how many HDL messages you can view during the Analysis & Elaboration of your design files. You can set the HDL Message Level to enable or disable groups of HDL messages, or you can enable or disable specific messages.

Related Information

- Synthesis Directives on page 16-25
  For more information about synthesis directives and their syntax

Setting the HDL Message Level

The HDL Message Level specifies the types of messages that the Quartus II software displays when it is analyzing and elaborating your design files.

Table 16-61: HDL Info Message Level

Level	Purpose	Description
Level1	High-severity messages only	If you want to view only the HDL messages that identify likely problems with your design, select Level1. When you select Level1, the Quartus II software issues a message only if there is an actual problem with your design.
Level2	High-severity and medium-severity messages	If you want to view additional HDL messages that identify possible problems with your design, select Level2. Level2 is the default setting.
Level3	All messages, including low-severity messages	If you want to view all HDL info and warning messages, select Level3. This level includes extra “LINT” messages that suggest changes to improve the style of your HDL code.

You must address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the Quartus II software, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**
2. Set the necessary message level from the pull-down menu in the **HDL Message Level** list, and then click **OK.**

   You can override this default setting in a source file with the `message_level synthesis` directive, which takes the values `level1`, `level2`, and `level3`, as shown in the following table.

Table 16-62: HDL Examples of message_level Directive

HDL	Code

Verilog HDL	`// altera message_level level1`
	`or`
	`/* altera message_level level3 */`

VHDL	`-- altera message_level level2`

A `message_level` synthesis directive remains effective until the end of a file or until the next `message_level` directive. In VHDL, you can use the `message_level` synthesis directive to set the HDL Message Level for entities and architectures, but not for other design units. An HDL Message Level for an entity applies to its architectures, unless overridden by another `message_level` directive. In Verilog HDL, you can use the `message_level` directive to set the HDL Message Level for a module.

Enabling or Disabling Specific HDL Messages by Module/Entity

Message ID is in parentheses at the beginning of the message. Use the Message ID to enable or disable a specific HDL info or warning message. Enabling or disabling a specific message overrides its HDL Message Level. This method is different from the message suppression in the Messages window because you can disable messages for a specific module or a specific entity. This method applies only to the HDL messages, and if you disable a message with this method, the Quartus II software lists the message as a suppressed message.

To disable specific HDL messages in the Quartus II software, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.
2. In the **Advanced Message Settings** dialog box, add the Message IDs you want to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. These directives require a space-separated list of Message IDs. You can enable or disable messages with these synthesis directives immediately before Verilog HDL modules, VHDL entities, or VHDL architectures. You cannot enable or disable a message during an HDL construct.

A message enabled or disabled via a `message_on` or `message_off` synthesis directive overrides its HDL Message Level or any `message_level` synthesis directive. The message remains disabled until the end of the source file or until you use another `message_on` or `message_off` directive to change the status of the message.

Table 16-63: HDL message_off Directive for Message with ID 10000

HDL	Code

Verilog HDL	`// altera message_off 10000`
	`or`
	`/* altera message_off 10000 */`

VHDL	`-- altera message_off 10000`

Node-Naming Conventions in Quartus II Integrated Synthesis

Whenever possible, Quartus II Integrated Synthesis uses wire or signal names from your source code to name nodes such as LEs or ALMs. Some nodes, such as registers, have predictable names that do not change when a design is resynthesized, although certain optimizations can affect register names. The names of other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.

Hierarchical Node-Naming Conventions

To make each name in your design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator indicates a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, with the ":" separator between each entity name and its instance name. For example, if a design defines entity `A` with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. You can obtain the full name of any node by starting with the hierarchical instance path, followed by a "|", and ending with the node name inside that entity.

This example shows you the convention:

```
<entity 0>:<instance_name 0>|<entity 1>:<instance_name 1>|...|<instance_name n>|
<node_name>
```

For example, if entity A contains a register (DFF atom) called `my_dff`, its full hierarchy name would be `A:my_A_inst|my_dff`.

To instruct the Compiler to generate node names that do not contain entity names, on the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings**, and then turn off D**isplay entity name for node name**.

With this option turned off, the node names use the convention in shown in this example:

```
<instance_name 0>|<instance_name 1>|...|<instance_name n> |<node_name>
```

Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)

In Verilog HDL and VHDL, inferred registers use the names of the `reg` or `signal` connected to the output.

Table 16-64: HDL Example of a Register that Creates `my_dff_out` DFF Primitive

HDL Register	Code

Verilog HDL	`wire dff_in, my_dff_out, clk;`
	`always @ (posedge clk)`
	`my_dff_out <= dff_in;`

HDL Register	Code

VHDL	
	```
	signal dff_in, my_dff_out, clk;
	process (clk)
	begin
	if (rising_edge(clk)) then
	my_dff_out <= dff_in;
	end if;
	end process;
	```

AHDL designs explicitly declare DFF registers rather than infer, so the software uses the user-declared name for the register.

For schematic designs using a **.bdf**, your design names all elements when you instantiate the elements in your design, so the software uses the name you defined for the register or DFF.

In the special case that a wire or signal (such as `my_dff_out` in the preceding examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (for example, cannot use `my_dff_out`) because the software requires that all logic and I/O cells have unique names. Here, Quartus II Integrated Synthesis appends `~reg0` to the register name.

Table 16-65: Verilog HDL Register Feeding Output Pin

For example, the Verilog HDL code example in this table generates a register called `q~reg0`.

HDL	Code

Verilog HDL	
	```
	module my_dff (input clk, input d, output q);
	always @ (posedge clk)
	q <= d;
	endmodule
	```

This situation occurs only for registers driving top-level pins. If a register drives a port of a lower level of the hierarchy, the software removes the port during hierarchy flattening and the register retains its original name, in this case, `q`.

Register Changes During Synthesis

On some occasions, you might not find registers that you expect to view in the synthesis netlist. Logic optimization might remove registers and synthesis optimizations might change the names of the registers. Common optimizations include inference of a state machine, counter, adder-subtractor, or shift register from registers and surrounding logic. Other common register changes occur when the software packs these registers into dedicated hardware on the FPGA, such as a DSP block or a RAM block.

The following factors can affect register names:

- Synthesis and Fitting Optimizations on page 16-72
- State Machines on page 16-72
- Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions on page 16-73
- Packed Input and Output Registers of RAM and DSP Blocks on page 16-73

Synthesis and Fitting Optimizations

Logic optimization during synthesis might remove registers if you do not connect the registers to inputs or outputs in your design, or if you can simplify the logic due to constant signal values. Synthesis optimizations might change register names, such as when the software merges duplicate registers to reduce resource utilization.

NOT-gate push back optimizations can affect registers that use preset signals. This type of optimization can impact your timing assignments when the software uses registers as clock dividers. If this situation occurs in your design, change the clock settings to work on the new register name.

Synthesis netlist optimizations often change node names because the software can combine or duplicate registers to optimize your design.

The Quartus II Compilation Report provides a list of registers that synthesis optimizations remove, and a brief reason for the removal. To generate the Quartus II Compilation Report, follow these steps:

1. In the **Analysis & Synthesis** folder, open **Optimization Results.**
2. Open **Register Statistics,** and then click the **Registers Removed During Synthesis** report.
3. Click **Removed Registers Triggering Further Register Optimizations**.

The second report contains a list of registers that causes synthesis optimizations to remove other registers from your design. The report provides a brief reason for the removal, and a list of registers that synthesis optimizations remove due to the removal of the initial register.

Quartus II Integrated Synthesis creates synonyms for registers duplicated with the **Maximum Fan-Out** option (or `maxfan` attribute). Therefore, timing assignments applied to nodes that are duplicated with this option are applied to the new nodes as well.

The Quartus II Fitter can also change node names after synthesis (for example, when the Fitter uses register packing to pack a register into an I/O element, or when physical synthesis modifies logic). The Fitter creates synonyms for duplicated registers so timing analysis can use the existing node name when applying assignments.

You can instruct the Quartus II software to preserve certain nodes throughout compilation so you can use them for verification or making assignments.

Related Information

Netlist Optimizations and Physical Synthesis
For more information about the type of optimizations performed by synthesis netlist optimizations

Preserving Register Names on page 16-73
For more information about preserving certain nodes throughout compilation

State Machines

If your HDL code infers a state machine, the software maps the registers that represent the states into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form in which one register represents each state. In this case, for Verilog HDL or VHDL designs, the registers take the name of the state register and the states.

For example, consider a Verilog HDL state machine in which the states are `parameter state0 = 1, state1 = 2`, `state2 = 3`, and in which the software declares the state machine register as `reg [1:0] my_fsm`. In this example, the three one-hot state registers are `my_fsm.state0`, `my_fsm.state1`, and `my_fsm.state2`.

An AHDL design explicitly specifies state machines with a state machine name. Your design names state machine registers with synthesized names based on the state machine name, but not the state names. For example, if a `my_fsm` state machine has four state bits, The software might synthesize these state bits with names such as `my_fsm~12`, `my_fsm~13`, `my_fsm~14`, and `my_fsm~15`.

Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions

The Quartus II software infers IP cores from Verilog HDL and VHDL code for logic that forms adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that are placed in DSP blocks.

Because adder-subtractors are part of an IP core instead of generic logic, the combinational logic exists in the design with different names. For shift registers, memory, and DSP functions, the software implements the registers and logic inside the dedicated RAM or DSP blocks in the device. Thus, the registers are not visible as separate LEs or ALMs.

Related Information

- **Recommended HDL Coding Styles** on page 12-1
  For information about inferring IP cores

Packed Input and Output Registers of RAM and DSP Blocks

The software packs registers into the input registers and output registers of RAM and DSP blocks, so that they are not visible as separate registers in LEs or ALMs.

Related Information

- **Recommended HDL Coding Styles** on page 12-1
  For information about packing registers into RAM and DSP IP cores

Preserving Register Names

Altera recommends that you preserve certain register names for verification or debugging, or to ensure that you applied timing assignments correctly. Quartus II Integrated Synthesis preserves certain nodes automatically if the software uses the nodes in a timing constraint.

Related Information

- **Preserve Registers** on page 16-38
  Use the `preserve` attribute to instruct the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations
- **Noprune Synthesis Attribute/Preserve Fan-out Free Register Node** on page 16-39
  Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow
- **Disable Register Merging/Don't Merge Register** on page 16-39
  Use the synthesis attribute `syn_dont_merge` to ensure that the Compiler does not merge registers with other registers

Node-Naming Conventions for Combinational Logic Cells

Whenever possible for Verilog HDL, VHDL, and AHDL code, the Quartus II software uses wire names that are the targets of assignments, but can change the node names due to synthesis optimizations.

For example, consider the Verilog HDL code in this example. Quartus II Integrated Synthesis uses the names `c`, `d`, `e`, and `f` for the generated combinational logic cells.

```
wire c;
reg d, e, f;
assign c = a | b;
always @ (a or b)
d = a & b;
always @ (a or b) begin : my_label
e = a ^ b;
end
always @ (a or b)
f = ~(a | b);
```

For schematic designs using a **.bdf**, your design names all elements when you instantiate the elements in your design and the software uses the name you defined when possible.

If logic cells are packed with registers in device architectures such as the Stratix and Cyclone device families, those names might not appear in the netlist after fitting. In other devices, such as newer families in the Stratix and Cyclone series device families, the register and combinational nodes are kept separate throughout the compilation, so these names are more often maintained through fitting.

When logic optimizations occur during synthesis, it is not always possible to retain the initial names as described. Sometimes, synthesized names are used, which are the wire names with a tilde (~) and a number appended. For example, if a complex expression is assigned to wire `w` and that expression generates several logic cells, those cells can have names such as `w`, `w~1`, and `w~2`. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. Quartus II Integrated Synthesis attempts to retain user names whenever possible. Any node name ending with ~<*number*> is a name created during synthesis, which can change if the design is changed and re-synthesized. Knowing these naming conventions helps you understand your post-synthesis results, helping you to debug your design or create assignments.

During synthesis, the software maintains combinational clock logic by not changing nodes that might be clocks. The software also maintains or protects multiplexers in clock trees, so that the TimeQuest analyzer has information about which paths are unate, to allow complete and correct analysis of combinational clocks. Multiplexers often occur in clock trees when the software selects between different clocks. To help with the analysis of clock trees, the software ensures that each multiplexer encountered in a clock tree is broken into 2:1 multiplexers, and each of those 2:1 multiplexers is mapped into one lookup table (independent of the device family). This optimization might result in a slight increase in area, and for some designs a decrease in timing performance. To disable the option, click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** > **Clock MUX Protection**.

Related Information

Clock MUX Protection logic option

For more information about Clock MUX Protection logic option and a list of supported devices

Preserving Combinational Logic Names

You can preserve certain combinational logic node names for verification or debugging, or to ensure that timing assignments are applied correctly.

Use the `keep` attribute to keep a wire name or combinational node name through logic synthesis minimizations and netlist optimizations.

For any internal node in your design clock network, use `keep` to protect the name so that you can apply correct clock settings. Also, set the attribute for combinational logic involved in `cut` and `-through` assignments.

Note: Setting the `keep` attribute for combinational logic can increase the area utilization and increase the delay of the final mapped logic because the attribute requires the insertion of extra combinational logic. Use the attribute only when necessary.

Related Information

- **Keep Combinational Node/Implement as Output of Logic Cell** on page 16-40

Scripting Support

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

To run the Help browser, type the command at the command prompt shown in this example:

```
quartus_sh --qhelp
```

You can specify many of the options either on an instance, at the global level, or both.

To make a global assignment, use the Tcl command shown in this example:

```
set_global_assignment -name <QSF Variable Name> <Value>
```

To make an instance assignment, use the Tcl command shown in this example:

```
set_instance_assignment -name <QSF Variable Name> <Value>\ -to <Instance Name>
```

To set the **Synthesis Effort** option at the command line, use the `--effort` option with the `quartus_map` executable shown in this example:

```
quartus_map <Design name> --effort= "auto | fast"
```

Related Information

- **Tcl Scripting**
  For more information about Tcl scripting
- **Quartus II Settings File Manual**
  For more information about all settings and constraints in the Quartus II software
- **Command-Line Scripting**
  For more information about command-line scripting
- **API Functions for Tcl**
  For more information about Tcl scripting
- **Synthesis Effort** on page 16-31

Adding an HDL File to a Project and Setting the HDL Version

To add an HDL or schematic entry design file to your project, use the Tcl assignments shown in this example:

```
set_global_assignment –name VERILOG_FILE <file name>.<v|sv>
set_global_assignment –name SYSTEMVERILOG_FILE <file name>.sv
set_global_assignment –name VHDL_FILE <file name>.<vhd|vhdl>
set_global_assignment -name AHDL_FILE <file name>.tdf
set_global_assignment -name BDF_FILE <file name>.bdf
```

Note: You can use any file extension for design files, as long as you specify the correct language when adding the design file. For example, you can use **.h** for Verilog HDL header files.

To specify the Verilog HDL or VHDL version, use the option shown in this example, at the end of the `VERILOG_FILE` or `VHDL_FILE` command:

```
-    HDL_VERSION <language version>
```

The variable <language version> takes one of the following values:

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`
- `VHDL_1987`
- `VHDL_1993`
- `VHDL_2008`

For example, to add a Verilog HDL file called **my_file.v** written in Verilog-1995, use the command shown in this example:

```
set_global_assignment –name VERILOG_FILE my_file.v –HDL_VERSION \ VERILOG_1995
```

In this example, the `syn_encoding` attribute associates a binary encoding with the states in the enumerated type `count_state`. In this example, the states are encoded with the following values: zero = "`11`", one = "`01`", two = "`10`", three = "`00`".

```
ARCHITECTURE rtl OF my_fsm IS
    TYPE count_state is (zero, one, two, three);
    ATTRIBUTE syn_encoding : STRING;
    ATTRIBUTE syn_encoding OF count_state : TYPE IS "11 01 10 00";
    SIGNAL present_state, next_state : count_state;
BEGIN
```

You can also use the `syn_encoding` attribute in Verilog HDL to direct the synthesis tool to use the encoding from your HDL code, instead of using the **State Machine Processing** option.

The `syn_encoding` value "`user`" instructs the Quartus II software to encode each state with its corresponding value from the Verilog HDL source code. By changing the values of your state constants, you can change the encoding of your state machine.

In **Example 16-8**, the states are encoded as follows:

```
init = "00"
last = "11"
```

```
next = "01"
later = "10"
```

Example 16-8: Verilog-2001 and SystemVerilog Code: Specifying User-Encoded States with the syn_encoding Attribute

```
(* syn_encoding = "user" *) reg [1:0] state;
parameter init = 0, last = 3, next = 1, later = 2;
always @ (state) begin
case (state)
init:
out = 2'b01;
next:
out = 2'b10;
later:
out = 2'b11;
last:
out = 2'b00;
endcase
end
```

Without the `syn_encoding` attribute, the Quartus II software encodes the state machine based on the current value of the **State Machine Processing** logic option.

If you also specify a safe state machine (as described in **Safe State Machine** on page 16-35), separate the encoding style value in the quotation marks from the safe value with a comma, as follows: "`safe, one-hot`" or "`safe, gray`".

Related Information

- **Safe State Machine** on page 16-35
- **Manually Specifying State Assignments Using the syn_encoding Attribute** on page 16-32

Assigning a Pin

To assign a signal to a pin or device location, use the Tcl command shown in this example:

```
set_location_assignment -to <signal name> <location>
```

Valid locations are pin location names. Some device families also support edge and I/O bank locations. Edge locations are `EDGE_BOTTOM`, `EDGE_LEFT`, `EDGE_TOP`, and `EDGE_RIGHT`. I/O bank locations include `IOBANK_1` to `IOBANK_n`, where `n` is the number of I/O banks in a device.

Creating Design Partitions for Incremental Compilation

To create a partition, use the command shown in this example:

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

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

The <*destination*> is the short hierarchy path of the entity. A short hierarchy path is the full hierarchy path without the top-level name, for example: `"ram:ram_unit|altsyncram:altsyncram_component"` (with quotation marks). For the top-level partition, you can use the pipe (`|`) symbol to represent the top-level entity.

The <*partition name*> is the partition name you designate, which should be unique and less than 1024 characters long. The name may only consist of alphanumeric characters, as well as pipe (`|`), colon (`:`), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (" ").

Related Information

- **Node-Naming Conventions in Quartus II Integrated Synthesis** on page 16-70
  For more information about hierarchical naming conventions

Quartus II Synthesis Options

Related Information

Logic options
For more information about the .qsf variable names and applicable values for the settings

Document Revision History

Table 16-66: Document Revision History

Date	Version	Changes
2015.05.04	15.0.0	• Removed support for early timing estimate feature. • Removed the note on the assignment of the RAM style attributes as it is no longer relevant.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.
2014.06.30	14.0.0	Template update.
November 2013	13.1.0	• Added a note regarding ROM inference using the `ram_init_file` in “RAM Initialization File—for Inferred Memory” on page 16–61.
May 2013	13.0.0	• Added “Verilog HDL Configuration” on page 16–6. • Added “RAM Style Attribute—For Shift Registers Inference” on page 16–57. • Added “Upgrade IP Components Dialog Box” on page 16–75.
June 2012	12.0.0	• Updated “Design Flow” on page 16–2.

Date	Version	Changes
November 2011	11.1.0	• Updated “Language Support” on page 16–5, “Incremental Compilation” on page 16–22, “Quartus II Synthesis Options” on page 16–24.
May 2011	11.0.0	• Updated “Specifying Pin Locations with chip_pin” on page 14–65, and “Shift Registers” on page 14–48. • Added a link to Quartus II Help in “SystemVerilog Support” on page 14–5. • Added Example 14–106 and Example 14–107 on page 14–67.
December 2010	10.1.0	• Updated “Verilog HDL Support” on page 13–4 to include Verilog-2001 support. • Updated “VHDL-2008 Support” on page 13–9 to include the condition operator (explicit and implicit) support. • Rewrote “Limiting Resource Usage in Partitions” on page 13–32. • Added “Creating LogicLock Regions” on page 13–32 and “Using Assignments to Limit the Number of RAM and DSP Blocks” on page 13–33. • Updated “Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute” on page 13–55. • Updated “Auto Gated Clock Conversion” on page 13–28. • Added links to Quartus II Help.

Date	Version	Changes
July 2010	10.0.0	• Removed Referenced Documents section. • Added "Synthesis Seed" on page 9–36 section. • Updated the following sections: "SystemVerilog Support" on page 9–5 "VHDL-2008 Support" on page 9–10 "Using Parameters/Generics" on page 9–16 "Parallel Synthesis" on page 9–21 "Limiting Resource Usage in Partitions" on page 9–32 "Synthesis Effort" on page 9–35 "Synthesis Attributes" on page 9–25 "Synthesis Directives" on page 9–27 "Auto Gated Clock Conversion" on page 9–29 "State Machine Processing" on page 9–36 "Multiply-Accumulators and Multiply-Adders" on page 9–50 "Resource Aware RAM, ROM, and Shift-Register Inference" on page 9–52 "RAM Style and ROM Style—for Inferred Memory" on page 9–53 "Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute" on page 9–55 "Using altera_attribute to Set Quartus II Logic Options" on page 9–68 "Adding an HDL File to a Project and Setting the HDL Version" on page 9–83 "Creating Design Partitions for Incremental Compilation" on page 9–85 "Inferring Multiplier, DSP, and Memory Functions from HDL Code" on page 9–50 • Updated Table 9–9 on page 9–86.
December 2009	9.1.1	• Added information clarifying inheritance of Synthesis settings by lower-level entities, including Altera and third-party IP • Updated "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46

Date	Version	Changes
November 2009	9.1.0	• Updated the following sections: "Initial Constructs and Memory System Tasks" on page 9–7 "VHDL Support" on page 9–9 "Parallel Synthesis" on page 9–21 "Synthesis Directives" on page 9–27 "Timing-Driven Synthesis" on page 9–31 "Safe State Machines" on page 9–40 "RAM Style and ROM Style—for Inferred Memory" on page 9–53 "Translate Off and On / Synthesis Off and On" on page 9–62 "Read Comments as HDL" on page 9–63 "Adding an HDL File to a Project and Setting the HDL Version" on page 9–81 • Removed "Remove Redundant Logic Cells" section • Added "Resource Aware RAM, ROM, and Shift-Register Inference" section • Updated Table 9–9 on page 9–83
March 2009	9.0.0	• Updated Table 9–9. • Updated the following sections: "Partitions for Preserving Hierarchical Boundaries" on page 9–20 "Analysis & Synthesis Settings Page of the Settings Dialog Box" on page 9–24 "Timing-Driven Synthesis" on page 9–30 "Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting" on page 9–54 • Added "Parallel Synthesis" on page 9–21 • Chapter 9 was previously Chapter 8 in software version 8.1

Related Information

- **Quartus II Handbook Archive**
  For previous versions of the Quartus II Handbook

Synopsys Synplify Support 17

2014.12.15

QII5V1 Subscribe Send Feedback

About Synplify Support

This manual delineates the support for the Synopsys Synplify software in the Quartus® II software, as well as key design flows, methodologies, and techniques for achieving optimal results in Altera® devices. The content in this manual applies to the Synplify, Synplify Pro, and Synplify Premier software unless otherwise specified. This manual assumes that you have set up, licensed, and are familiar with the Synplify software.

This manual includes the following information:

- General design flow with the Synplify and Quartus II software
- Exporting designs and constraints to the Quartus II software using NativeLink integration
- Synplify software optimization strategies, including timing-driven compilation settings, optimization options, and Altera-specific attributes
- Guidelines for Altera IP cores and library of parameterized module (LPM) functions, instantiating them with the IP Catalog, and tips for inferring them from hardware description language (HDL) code
- Incremental compilation and block-based design, including the MultiPoint flow in the Synplify Pro and Synplify Premier software

Related Information

- **Synplify Synthesis Techniques with the Quartus II Software online training**
- **Synplify Pro Tips and Tricks online training**

Design Flow

The following steps describe a basic Quartus II software design flow using the Synplify software:

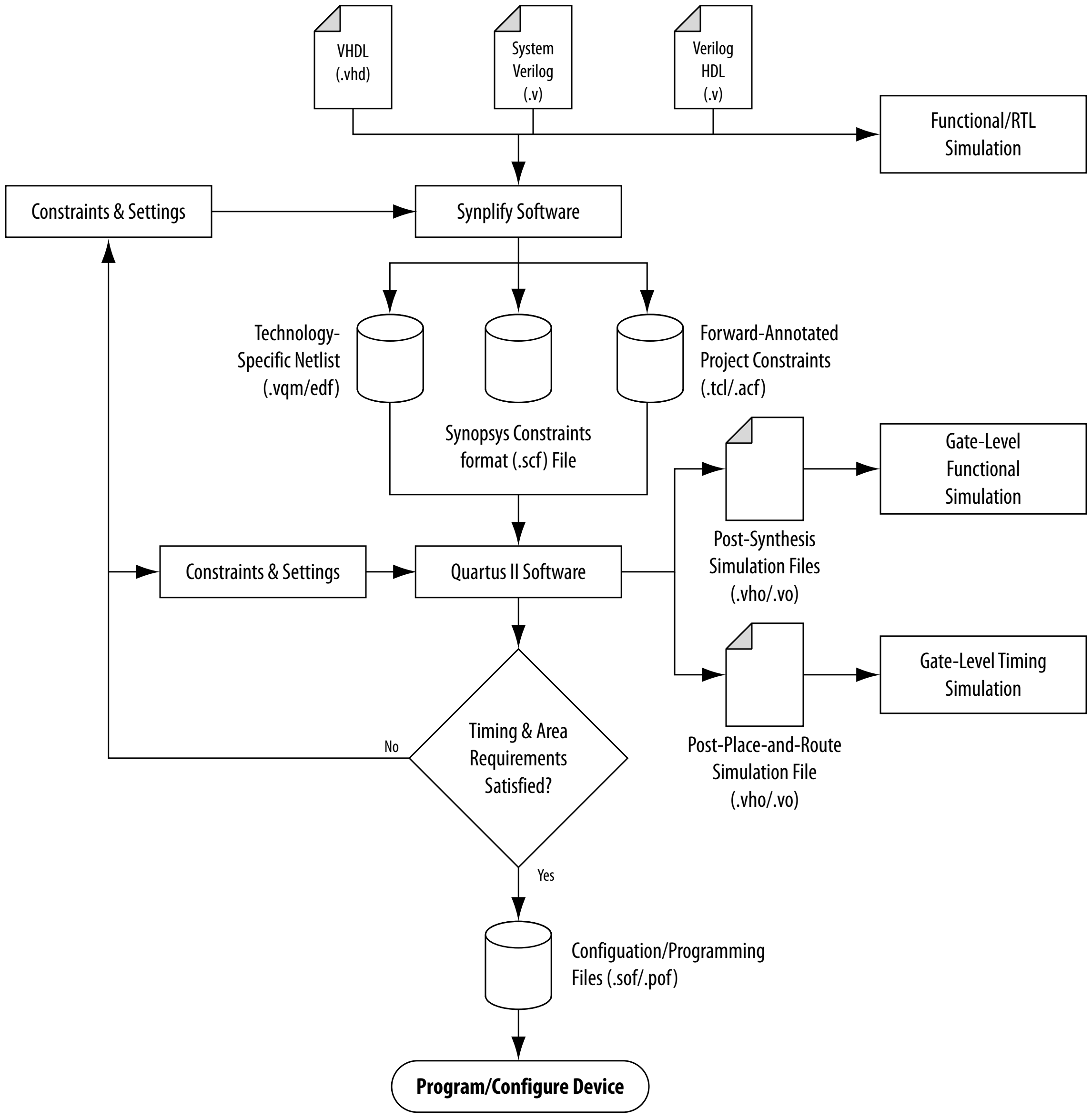
1. Create Verilog HDL or VHDL design files.
2. Set up a project in the Synplify software and add the HDL design files for synthesis.
3. Select a target device and add timing constraints and compiler directives in the Synplify software to help optimize the design during synthesis.
4. Synthesize the project in the Synplify software.
5. Create a Quartus II project and import the following files generated by the Synplify software into the Quartus II software. Use the following files for placement and routing, and for performance evaluation:

- Verilog Quartus Mapping File (**.vqm**) netlist.
- The Synopsys Constraints Format (**.scf**) file for TimeQuest Timing Analyzer constraints.
- The **.tcl** file to set up your Quartus II project and pass constraints.

Note: Alternatively, you can run the Quartus II software from within the Synplify software.

6. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 17-1: Recommended Design Flow


VHDL (.vhd)
System Verilog (.v)
Verilog HDL (.v)
Functional/RTL Simulation
Constraints & Settings
Synplify Software
Technology-Specific Netlist (.vqm/edf)
Synopsys Constraints format (.scf) File
Forward-Annotated Project Constraints (.tcl/.acf)
Gate-Level Functional Simulation
Post-Synthesis Simulation Files (.vho/.vo)
Constraints & Settings
Quartus II Software
Gate-Level Timing Simulation
Post-Place-and-Route Simulation File (.vho/.vo)
No
Timing & Area Requirements Satisfied?
Yes
Configuation/Programming Files (.sof/.pof)
Program/Configure Device

Related Information

- **Running the Quartus II Software from within the Synplify Software** on page 17-4
- **Synplify Software Generated Files** on page 17-5
- **Design Constraints Support** on page 17-6

Hardware Description Language Support

The Synplify software supports VHDL, Verilog HDL, and SystemVerilog source files. However, only the Synplify Pro and Premier software support mixed synthesis, allowing a combination of VHDL and Verilog HDL or SystemVerilog format source files.

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent RTL view netlist (**.srs**) and technology-view netlist (**.srm**) files. You can use the Synplify HDL Analyst to analyze and debug your design visually. The HDL Analyst supports cross-probing between the RTL and Technology views, the HDL source code, the Finite State Machine (FSM) viewer, and between the technology view and the timing report file in the Quartus II software. A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro and Premier software include the HDL Analyst.

Related Information

Guidelines for Altera IP Cores and Architecture-Specific Features on page 17-15

Altera Device Family Support

Support for newly released device families may require an overlay. Contact Synopsys for more information.

Related Information

Synopsys Website

Tool Setup

Specifying the Quartus II Software Version

You can specify your version of the Quartus II software in **Implementation Options** in the Synplify software. This option ensures that the netlist is compatible with the software version and supports the newest features. Altera recommends using the latest version of the Quartus II software whenever possible. If your Quartus II software version is newer than the versions available in the **Quartus Version** list, check if there is a newer version of the Synplify software available that supports the current Quartus II software version. Otherwise, select the latest version in the list for the best compatibility.

Note: The **Quartus Version** list is available only after selecting an Altera device.

Example 17-1: Specifying Quartus II Software Version at the Command Line

```
set_option -quartus_version <version number>
```

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, and allows you to run other EDA design entry or synthesis, simulation, and timing analysis tools automatically from within the Quartus II software. After a design is synthesized in the Synplify software, a **.vqm** netlist file, an **.scf** file for TimeQuest Timing Analyzer timing constraints, and **.tcl** files are used to import the design into the Quartus II software for place-and-route. You can run the Quartus II software from within the Synplify software or as a stand-alone application. After you import the design into the Quartus II software, you can specify different options to further optimize the design.

Note: When you are using NativeLink integration, the path to your project must not contain empty spaces. The Synplify software uses Tcl scripts to communicate with the Quartus II software, and the Tcl language does not accept arguments with empty spaces in the path.

Use NativeLink integration to integrate the Synplify software and Quartus II software with a single GUI for both synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus II software from within the Synplify software GUI, or to run the Synplify software from within the Quartus II software GUI.

Running the Quartus II Software from within the Synplify Software

To run the Quartus II software from within the Synplify software, you must set the *QUARTUS_ROOTDIR* environment variable to the Quartus II software installation directory located in <*Quartus II system directory*>\altera\ <*version number*>**\quartus**. You must set this environment variable to use the Synplify and Quartus II software together. Synplify also uses this variable to open the Quartus II software in the background and obtain detailed information about the Altera IP cores used in the design.

For the Windows operating system, do the following:

1. Point to **Start**, and click **Control Panel**.
2. Click **System** >**Advanced system settings** >**Environment Variables**.
3. Create a *QUARTUS_ROOTDIR* system variable.

For the Linux operating system, do the following:

- Create an environment variable *QUARTUS_ROOTDIR* that points to the <*home directory*>/altera <*version number*> location.

You can create new place and route implementations with the **New P&R** button in the Synplify software GUI. Under each implementation, the Synplify Pro software creates a place-and-route implementation called **pr_**<*number*> **Altera Place and Route**. To run the Quartus II software in command-line mode after each synthesis run, use the text box to turn on the place-and-route implementation. The results of the place-and-route are written to a log file in the **pr_** <*number*> directory under the current implementation directory.

You can also use the commands in the Quartus II menu to run the Quartus II software at any time following a successful completion of synthesis. In the Synplify software, on the Options menu, click **Quartus II** and then choose one of the following commands:

- **Launch Quartus** —Opens the Quartus II software GUI and creates a Quartus II project with the synthesized output file, forward-annotated timing constraints, and pin assignments. Use this command to configure options for the project and to execute any Quartus II commands.
- **Run Background Compile**—Runs the Quartus II software in command-line mode with the project settings from the synthesis run. The results of the place-and-route are written to a log file.

The <*project_name*>**_cons.tcl** file is used to set up the Quartus II project and directs the <*project_name*>**.tcl** file to pass constraints from the Synplify software to the Quartus II software. By default, the <*project_name*>**.tcl** file contains device, timing, and location assignments. The

<project_name>.**tcl** file contains the command to use the Synplify-generated **.scf** constraints file with the TimeQuest Timing Analyzer.

Related Information

Design Flow on page 17-1

Using the Quartus II Software to Run the Synplify Software

You can set up the Quartus II software to run the Synplify software for synthesis with NativeLink integration. This feature allows you to use the Synplify software to quickly synthesize a design as part of a standard compilation in the Quartus II software. When you use this feature, the Synplify software does not use any timing constraints or assignments, such as incremental compilation partitions, that you have set in the Quartus II software.

Note: For best results, Synopsys recommends that you set constraints in the Synplify software and use a Tcl script to pass these constraints to the Quartus II software, instead of opening the Synplify software from within the Quartus II software.

To set up the Quartus II software to run the Synplify software, do the following:

1. On the Tools menu, click **Options**.
2. In the **Options** dialog box, click **EDA Tool Options** and specify the path of the Synplify or Synplify Pro software under **Location of Executable**.

Running the Synplify software with NativeLink integration is supported on both floating network and node-locked fixed PC licenses. Both types of licenses support batch mode compilation.

Related Information

About Using the Synplify Software with the Quartus II Software Online Help

Synplify Software Generated Files

During synthesis, the Synplify software produces several intermediate and output files.

Table 17-1: Synplify Intermediate and Output Files

File Extensions	File Description
.vqm	Technology-specific netlist in **.vqm** file format. A **.vqm** file is created for all Altera device families supported by the Quartus II software.
.scf(13)	Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer.

(13) If your design uses the Classic Timing Analyzer for timing analysis in the Quartus II software versions 10.0 and earlier, the Synplify software generates timing constraints in the Tcl Constraints File (**.tcl**). If you are using the Quartus II software versions 10.1 and later, you must use the TimeQuest Timing Analyzer for timing analysis.

File Extensions	File Description
.tcl	Forward-annotated constraints file containing constraints and assignments. A **.tcl** file for the Quartus II software is created for all devices. The **.tcl** file contains the appropriate Tcl commands to create and set up a Quartus II project and pass placement constraints.
.srs	Technology-independent RTL netlist file that can be read only by the Synplify software.
.srm	Technology view netlist file.
.acf	Assignment and Configurations file for backward compatibility with the MAX +PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II **.acf** file.
.srr[14]	Synthesis Report file.

Related Information

Design Flow on page 17-1

Design Constraints Support

You can specify timing constraints and attributes by using the SCOPE window of the Synplify software, by editing the **.sdc** file, or by defining the compiler directives in the HDL source file. The Synplify software forward-annotates many of these constraints to the Quartus II software.

After synthesis is complete, do the following steps:

1. Import the **.vqm** netlist to the Quartus II software for place-and-route.
2. Use the **.tcl** file generated by the Synplify software to forward-annotate your project constraints including device selection. The **.tcl** file calls the generated **.scf** to foward-annotate TimeQuest Timing Analyzer timing constraints.

Related Information

- Design Flow on page 17-1
- Synplify Optimization Strategies on page 17-8
- **Netlist Optimizations and Physical Synthesis Documentation**

(14) This report file includes performance estimates that are often based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route—it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics that might inaccurately predict resource usage after place-and-route. The Synplify software does not account for black box functions nor for logic usage reduction achieved through register packing performed by the Quartus II software. Register packing combines a single register and look-up table (LUT) into a single logic cell, reducing logic cell utilization below the Synplify software estimate. Use the device utilization reported by the Quartus II software after place-and-route.

Running the Quartus II Software Manually With the Synplify-Generated Tcl Script

You can run the Quartus II software with a Synplify-generated Tcl script.

To run the Tcl script to set up your project assignments, perform the following steps:

1. Ensure the **.vqm**, **.scf**, and **.tcl** files are located in the same directory.
2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console opens.
3. At the Tcl Console command prompt, type the following:

```
source <path>/<project name>_cons.tcl
```

Passing TimeQuest SDC Timing Constraints to the Quartus II Software

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry standard constraints format, Synopsys Design Constraints (SDC).

The Synplify-generated **.tcl** file contains constraints for the Quartus II software, such as the device specification and any location constraints. Timing constraints are forward-annotated in the Synopsys Constraints Format (**.scf**) file.

Note: Synopsys recommends that you modify constraints using the SCOPE constraint editor window, rather than using the generated **.sdc**, **.scf**, or **.tcl** file.

The following list of Synplify constraints are converted to the equivalent Quartus II SDC commands and are forward-annotated to the Quartus II software in the **.scf** file:

- `define_clock`
- `define_input_delay`
- `define_output_delay`
- `define_multicycle_path`
- `define_false_path`

All Synplify constraints described above are mapped to SDC commands for the TimeQuest Timing Analyzer.

For syntax and arguments for these commands, refer to the applicable topic in this manual or refer to Synplify Help. For a list of corresponding commands in the Quartus II software, refer to the Quartus II Help.

Related Information

- **Timing-Driven Synthesis Settings** on page 17-9
- **Quartus II TimeQuest Timing Analyzer Documentation**
- **Quartus II Online Help**

Individual Clocks and Frequencies

Specify clock frequencies for individual clocks in the Synplify software with the `define_clock` command. This command is passed to the Quartus II software with the `create_clock` command.

Input and Output Delay

Specify input delay and output delay constraints in the Synplify software with the `define_input_delay` and `define_output_delay` commands, respectively. These commands are passed to the Quartus II software with the `set_input_delay` and `set_output_delay` commands.

Multicycle Path

Specify a multicycle path constraint in the Synplify software with the `define_multicycle_path` command. This command is passed to the Quartus II software with the `set_multicycle_path` command.

False Path

Specify a false path constraint in the Synplify software with the `define_false_path` command. This command is passed to the Quartus II software with the `set_false_path` command.

Simulation and Formal Verification

You can perform simulation and formal verification at various stages in the design process. You can perform final timing analysis after placement and routing is complete.

If area and timing requirements are satisfied, use the files generated by the Quartus II software to program or configure the Altera device. If your area or timing requirements are not met, you can change the constraints in the Synplify software or the Quartus II software and rerun synthesis. Altera recommends that you provide timing constraints in the Synplify software and any placement constraints in the Quartus II software. Repeat the process until area and timing requirements are met.

You can also use other options and techniques in the Quartus II software to meet area and timing requirements, such as WYSIWYG Primitive Resynthesis, which can perform optimizations on your **.vqm** netlist within the Quartus II software.

Note: In some cases, you might be required to modify the source code if the area and timing requirements cannot be met using options in the Synplify and Quartus II software.

Synplify Optimization Strategies

Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus II software options can help you obtain the results that you require.

For more information about applying attributes, refer to the *Synopsys FPGA Synthesis Reference Manual*.

Related Information

- **Design Constraints Support** on page 17-6
- **Recommended Design Practices Documentation** on page 11-1
- **Timing Closure and Optimization Documentation**

Using Synplify Premier to Optimize Your Design

Compared to other Synplify products, the Synplify Premier software offers additional physical synthesis optimizations. After typical logic synthesis, the Synplify Premier software places and routes the design and attempts to restructure the netlist based on the physical location of the logic in the Altera device. The

Synplify Premier software forward-annotates the design netlist to the Quartus II software to perform the final placement and routing. In the default flow, the Synplify Premier software also forward-annotates placement information for the critical path(s) in the design, which can improve the compilation time in the Quartus II software.

The physical location annotation file is called <*design name*>**_plc.tcl**. If you open the Quartus II software from the Synplify Premier software user interface, the Quartus II software automatically uses this file for the placement information.

The Physical Analyst allows you to examine the placed netlist from the Synplify Premier software, which is similar to the HDL Analyst for a logical netlist. You can use this display to analyze and diagnose potential problems.

Using Implementations in Synplify Pro or Premier

You can create different synthesis results without overwriting the existing results, in the Synplify Pro or Premier software, by creating a new implementation from the Project menu. For each implementation, specify the target device, synthesis options, and constraint files. Each implementation generates its own subdirectory that contains all the resulting files, including **.vqm**, **.scf**, and **.tcl** files, from a compilation of the particular implementation. You can then compare the results of the different implementations to find the optimal set of synthesis options and constraints for a design.

Timing-Driven Synthesis Settings

The Synplify software supports timing-driven synthesis with user-assigned timing constraints to optimize the performance of the design.

The Quartus II NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus II software with an **.scf** file for timing-driven place and route.

The Synplify Synthesis Report File (**.srr**) contains timing reports of estimated place-and-route delays. The Quartus II software can perform further optimizations on a post-synthesis netlist from third-party synthesis tools. In addition, designs might contain black boxes or intellectual property (IP) functions that have not been optimized by the third-party synthesis software. Actual timing results are obtained only after the design has been fully placed and routed in the Quartus II software. For these reasons, the Quartus II post place-and-route timing reports provide a more accurate representation of the design. Use the statistics in these reports to evaluate design performance.

Related Information

- **Passing TimeQuest SDC Timing Constraints to the Quartus II Software** on page 17-7
- **Exporting Designs to the Quartus II Software Using NativeLink Integration** on page 17-3

Clock Frequencies

For single-clock designs, you can specify a global frequency when using the push-button flow. While this flow is simple and provides good results, it often does not meet the performance requirements for more advanced designs. You can use timing constraints, compiler directives, and other attributes to help optimize the performance of a design. You can enter these attributes and directives directly in the HDL code. Alternatively, you can enter attributes (not directives) into an **.sdc** file with the SCOPE window in the Synplify software.

Use the SCOPE window to set global frequency requirements for the entire design and individual clock settings. Use the **Clocks** tab in the SCOPE window to specify frequency (or period), rise times, fall times, duty cycle, and other settings. Assigning individual clock settings, rather than over-constraining the global frequency, helps the Quartus II software and the Synplify software achieve the fastest clock frequency for the overall design. The `define_clock` attribute assigns clock constraints.

Multiple Clock Domains

The Synplify software can perform timing analysis on unrelated clock domains. Each clock group is a different clock domain and is treated as unrelated to the clocks in all other clock groups. All clocks in a single clock group are assumed to be related, and the Synplify software automatically calculates the relationship between the clocks. You can assign clocks to a new clock group or put related clocks in the same clock group with the **Clocks** tab in the SCOPE window, or with the `define_clock` attribute.

Input and Output Delays

Specify the input and output delays for the ports of a design in the **Input/Output** tab of the SCOPE window, or with the `define_input_delay` and `define_output_delay` attributes. The Synplify software does not allow you to assign the t_{CO} and t_{SU} values directly to inputs and outputs. However, a t_{CO} value can be inferred by setting an external output delay; a t_{SU} value can be inferred by setting an external input delay.

Relationship Between t_{CO} and the Output Delay

t_{CO} = clock period – external output delay

Relationship Between t_{SU} and the Input Delay

t_{SU} = clock period – external input delay

When the `syn_forward_io_constraints` attribute is set to 1, the Synplify software passes the external input and output delays to the Quartus II software using NativeLink integration. The Quartus II software then uses the external delays to calculate the maximum system frequency.

Multicycle Paths

A multicycle path is a path that requires more than one clock cycle to propagate. Specify any multicycle paths in the design in the **Multi-Cycle Paths** tab of the SCOPE window, or with the `define_multicycle_path` attribute. You should specify which paths are multicycle to prevent the Quartus II and the Synplify compilers from working excessively on a non-critical path. Not specifying these paths can also result in an inaccurate critical path reported during timing analysis.

False Paths

False paths are paths that should be ignored during timing analysis, or should be assigned low (or no) priority during optimization. Some examples of false paths include slow asynchronous resets, and test logic that has been added to the design. Set these paths in the **False Paths** tab of the SCOPE window, or use the `define_false_path` attribute.

FSM Compiler

If the FSM Compiler is turned on, the compiler automatically detects state machines in a design, which are then extracted and optimized. The FSM Compiler analyzes state machines and implements sequential, gray, or one-hot encoding, based on the number of states. The compiler also performs unused-state analysis, optimization of unreachable states, and minimization of transition logic. Implementation is based on the number of states, regardless of the coding style in the HDL code.

If the FSM Compiler is turned off, the compiler does not optimize logic as state machines. The state machines are implemented as HDL code. Thus, if the coding style for a state machine is sequential, the implementation is also sequential.

Use the `syn_state_machine` compiler directive to specify or prevent a state machine from being extracted and optimized. To override the default encoding of the FSM Compiler, use the `syn_encoding` directive.

Table 17-2: `syn_encoding` Directive Values

Value	Description
Sequential	Generates state machines with the fewest possible flipflops. Sequential, also called binary, state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitches.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines typically provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than sequential implementations.
Safe	Generates extra control logic to force the state machine to the reset state if an invalid state is reached. You can use the safe value in conjunction with any of the other three values, which results in the state machine being implemented with the requested encoding scheme and the generation of the reset logic.

Example 17-2: Sample VHDL Code for Applying `syn_encoding` Directive

```
SIGNAL current_state : STD_LOGIC_VECTOR (7 DOWNTO 0);
ATTRIBUTE syn_encoding : STRING;
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

By default, the state machine logic is optimized for speed and area, which may be potentially undesirable for critical systems. The safe value generates extra control logic to force the state machine to the reset state if an invalid state is reached.

FSM Explorer in Synplify Pro and Premier

The Synplify Pro and Premier software use the FSM Explorer to explore different encoding styles for a state machine automatically, and then implement the best encoding based on the overall design constraints. The FSM Explorer uses the FSM Compiler to identify and extract state machines from a design. However, unlike the FSM Compiler, which chooses the encoding style based on the number of states, the FSM Explorer attempts several different encoding styles before choosing a specific one. The trade-off is that the compilation requires more time to analyze the state machine, but finds an optimal encoding scheme for the state machine.

Optimization Attributes and Options

Retiming in Synplify Pro and Premier

The Synplify Pro and Premier software can retime a design, which can improve the timing performance of sequential circuits by moving registers (register balancing) across combinational elements. Be aware that retimed registers incur name changes. You can retime your design from **Implementation Options** or you can use the `syn_allow_retiming` attribute.

Maximum Fan-Out

When your design has critical path nets with high fan-out, use the `syn_maxfan` attribute to control the fan-out of the net. Setting this attribute for a specific net results in the replication of the driver of the net to reduce overall fan-out. The `syn_maxfan` attribute takes an integer value and applies it to inputs or registers. The `syn_maxfan` attribute cannot be used to duplicate control signals. The minimum allowed value of the attribute is 4. Using this attribute might result in increased logic resource utilization, thus straining routing resources, which can lead to long compilation times and difficult fitting.

If you must duplicate an output register or an output enable register, you can create a register for each output pin by using the `syn_useioff` attribute.

Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets cannot be maintained to create an optimized circuit. Applying the `syn_keep` directive overrides the optimization of the compiler and preserves the net during synthesis. The `syn_keep` directive is a Boolean data type value and can be applied to wires (Verilog HDL) and signals (VHDL). Setting the value to **true** preserves the net through synthesis.

Register Packing

Altera devices allow register packing into I/O cells. Altera recommends allowing the Quartus II software to make the I/O register assignments. However, you can control register packing with the `syn_useioff` attribute. The `syn_useioff` attribute is a Boolean data type value that can be applied to ports or entire modules. Setting the value to **1** instructs the compiler to pack the register into an I/O cell. Setting the value to **0** prevents register packing in both the Synplify and Quartus II software.

Resource Sharing

The Synplify software uses resource sharing techniques during synthesis, by default, to reduce area. Turning off the **Resource Sharing** option on the **Options** tab of the **Implementation Options** dialog box improves performance results for some designs. You can also turn off the option for a specific module with the `syn_sharing` attribute. If you turn off this option, be sure to check the results to verify improvement in timing performance. If there is no improvement, turn on **Resource Sharing**.

Preserving Hierarchy

The Synplify software performs cross-boundary optimization by default, which causes the design to flatten to allow optimization. You can use the `syn_hier` attribute to override the default compiler settings. The `syn_hier` attribute applies a string value to modules, architectures, or both. Setting the value to **hard** maintains the boundaries of a module, architecture, or both, but allows constant propagation. Setting the value to **locked** prevents all cross-boundary optimizations. Use the **locked** setting with the partition setting to create separate design blocks and multiple output netlists for incremental compilation.

By default, the Synplify software generates a hierarchical **.vqm** file. To flatten the file, set the `syn_netlist_hierarchy` attribute to **0**.

Related Information

Using MultiPoint Synthesis with Incremental Compilation on page 17-26

Register Input and Output Delays

Two advanced options, `define_reg_input_delay` and `define_reg_output_delay`, can speed up paths feeding a register, or coming from a register, by a specific number of nanoseconds. The Synplify software attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with the `define_clock` attribute). You can use these attributes to add a delay to paths feeding into or out of registers to further constrain critical paths. You can slow down a path that is too highly optimized by setting this attributes to a negative number.

The `define_reg_input_delay` and `define_reg_output_delay` options are useful to close timing if your design does not meet timing goals, because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. Rerun synthesis using these options, specifying the actual routing delay (from place-and-route results) so that the tool can meet the required clock frequency. Synopsys recommends that for best results, do not make these assignments too aggressively. For example, you can increase the routing delay value, but do not also use the full routing delay from the last compilation.

In the SCOPE constraint window, the registers panel contains the following options:

- **Register**—Specifies the name of the register. If you have initialized a compiled design, select the name from the list.
- **Type**—Specifies whether the delay is an input or output delay.
- **Route**—Shrinks the effective period for the constrained registers by the specified value without affecting the clock period that is forward-annotated to the Quartus II software.

Use the following Tcl command syntax to specify an input or output register delay in nanoseconds.

Example 17-3: Input and Output Register Delay

```
define_reg_input_delay {<register>} -route <delay in ns>
define_reg_output_delay {<register>} -route <delay in ns>
```

syn_direct_enable

This attribute controls the assignment of a clock-enable net to the dedicated enable pin of a register. With this attribute, you can direct the Synplify mapper to use a particular net as the only clock enable when the design has multiple clock enable candidates.

To use this attribute as a compiler directive to infer registers with clock enables, enter the `syn_direct_enable` directive in your source code, instead of the SCOPE spreadsheet.

The `syn_direct_enable` data type is Boolean. A value of **1** or **true** enables net assignment to the clock-enable pin. The following is the syntax for Verilog HDL:

```
object /* synthesis syn_direct_enable = 1 */ ;
```

I/O Standard

For certain Altera devices, specify the I/O standard type for an I/O pad in the design with the **I/O Standard** panel in the Synplify SCOPE window.

The Synplify SDC syntax for the `define_io_standard` constraint, in which the `delay_type` must be either `input_delay` or `output_delay`.

Example 17-4: define_io_standard Constraint

```
define_io_standard [-disable|-enable] {<objectName>} -delay_type \
[input_delay|output_delay] <columnTclName>{<value>}
[<columnTclName>{<value>}...]
```

For details about supported I/O standards, refer to the *Synopsys FPGA Synthesis Reference Manual.*

Altera-Specific Attributes

You can use the `altera_chip_pin_lc`, `altera_io_powerup`, and `altera_io_opendrain` attributes with specific Altera device features, which are forward-annotated to the Quartus II project, and are used during place-and-route.

altera_chip_pin_lc

Use the `altera_chip_pin_lc` attribute to make pin assignments. This attribute applies a string value to inputs and outputs. Use the attribute only on the ports of the top-level entity in the design. Do not use this attribute to assign pin locations from entities at lower levels of the design hierarchy.

Note: The `altera_chip_pin_lc` attribute is not supported for any MAX series device.

In the SCOPE window, set the value of the `altera_chip_pin_lc` attribute to a pin number or a list of pin numbers.

You can use VHDL code for making location assignments for supported Altera devices. Pin location assignments for these devices are written to the output **.tcl** file.

Note: The `data_out` signal is a 4-bit signal; `data_out[3]` is assigned to pin 14 and `data_out[0]` is assigned to pin 15.

Example 17-5: Making Location Assignments in VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "14, 5, 16, 15";
```

altera_io_powerup

Use the `altera_io_powerup` attribute to define the power-up value of an I/O register that has no set or reset. This attribute applies a string value (**high**|**low**) to ports with I/O registers. By default, the power-up value of the I/O register is set to **low**.

altera_io_opendrain

Use the `altera_io_opendrain` attribute to specify open-drain mode I/O ports. This attribute applies a boolean data type value to outputs or bidirectional ports for devices that support open-drain mode.

Guidelines for Altera IP Cores and Architecture-Specific Features

Altera provides parameterizable IP cores, including LPMs, device-specific Altera IP cores, and IP available through the Altera Megafunction Partners Program (AMPP^{SM}). You can use IP cores by instantiating them in your HDL code, or by inferring certain IP cores from generic HDL code.

You can instantiate an IP core in your HDL code with the IP Catalog and configure the IP core with the Parameter Editor, or instantiate the IP core using the port and parameter definition. The IP Catalog and Parameter Editor provide a graphical interface within the Quartus II software to customize any available Altera IP core for the design.

The Synplify software also automatically recognizes certain types of HDL code, and infers the appropriate Altera IP core when an IP core provides optimal results. The Synplify software provides options to control inference of certain types of IP cores.

Related Information

- **Hardware Description Language Support** on page 17-3
- **Recommended HDL Coding Styles Documentation** on page 12-1
- **About the IP Catalog Online Help**

Instantiating Altera IP Cores with the IP Catalog

When you use the IP Catalog and Parameter Editor to set up and configure an IP core, the IP Catalog creates a VHDL or Verilog HDL wrapper file <*output file*>**.v|vhd** that instantiates the IP core.

The Synplify software uses the Quartus II timing and resource estimation netlist feature to report more accurate resource utilization and timing performance estimates, and leverages timing-driven optimization, instead of treating the IP core as a "black box." Including the generated IP core variation wrapper file in your Synplify project, gives the Synplify software complete information about the IP core.

Note: There is an option in the Parameter Editor to generate a netlist for resource and timing estimation. This option is not recommended for the Synplify software because the software automatically generates this information in the background without a separate netlist. If you do create a separate netlist <*output file*>**_syn.v** and use that file in your synthesis project, you must also include the <*output file*>**.v|vhd** file in your Quartus II project.

Verify that the correct Quartus II version is specified in the Synplify software before compiling the generated file to ensure that the software uses the correct library definitions for the IP core. The **Quartus Version** setting must match the version of the Quartus II software used to generate the customized IP core.

In addition, ensure that the *QUARTUS_ROOTDIR* environment variable specifies the installation directory location of the correct Quartus II version. The Synplify software uses this information to launch the Quartus II software in the background. The environment variable setting must match the version of the Quartus II software used to generate the customized IP core.

Related Information

- Specifying the Quartus II Software Version on page 17-3
- Using the Quartus II Software to Run the Synplify Software on page 17-5

Instantiating Altera IP Cores with IP Catalog Generated Verilog HDL Files

If you turn on the <*output file*>**_inst.v** option on the Parameter Editor, the IP Catalog generates a Verilog HDL instantiation template file for use in your Synplify design. The instantiation template file, <*output file*>**_inst.v**, helps to instantiate the IP core variation wrapper file, <*output file*>**.v**, in your top-level design. Include the IP core variation wrapper file <*output file*>**.v** in your Synplify project. The Synplify software includes the IP core information in the output **.vqm** netlist file. You do not need to include the generated IP core variation wrapper file in your Quartus II project.

Instantiating Altera IP Cores with IP Catalog Generated VHDL Files

If you turn on the <*output file*>**.cmp** and <*output file*>**_inst.vhd** options on the Parameter Editor, the IP catalog generates a VHDL component declaration file and a VHDL instantiation template file for use in your Synplify design. These files can help you instantiate the IP core variation wrapper file, <*output file*>**.vhd**, in your top-level design. Include the <*output file*>**.vhd** in your Synplify project. The Synplify software includes the IP core information in the output **.vqm** netlist file. You do not need to include the generated IP core variation wrapper file in your Quartus II project.

Changing Synplify's Default Behavior for Instantiated Altera IP Cores

By default, the Synplify software automatically opens the Quartus II software in the background to generate a resource and timing estimation netlist for IP cores.

You might want to change this behavior to reduce run times in the Synplify software, because generating the netlist files can take several minutes for large designs, or if the Synplify software cannot access your Quartus II software installation to generate the files. Changing this behavior might speed up the compilation time in the Synplify software, but the Quality of Results (QoR) might be reduced.

The Synplify software directs the Quartus II software to generate information in two ways:

- Some IP cores provide a "clear box" model—the Synplify software fully synthesizes this model and includes the device architecture-specific primitives in the output **.vqm** netlist file.
- Other IP cores provide a "grey box" model—the Synplify software reads the resource information, but the netlist does not contain all the logic functionality.

  Note: You need to turn on **Generate netlist** when using the grey box model. For more information, see the Quartus II online help.

For these IP cores, the Synplify software uses the logic information for resource and timing estimation and optimization, and then instantiates the IP core in the output **.vqm** netlist file so the Quartus II software can implement the appropriate device primitives. By default, the Synplify software uses the clear box model when available, and otherwise uses the grey box model.

Related Information

- Including Files for Quartus II Placement and Routing Only on page 17-19
- Synplify Synthesis Techniques with the Quartus II Software online training
  Includes more information about design flows using clear box model and grey box model.
- Generating a Netlist for 3rd Party Synthesis Tools online help

Instantiating Intellectual Property with the IP Catalog and Parameter Editor

Many Altera IP cores include a resource and timing estimation netlist that the Synplify software uses to report more accurate resource utilization and timing performance estimates, and leverage timing-driven optimization rather than a black box function.

To create this netlist file, perform the following steps:

1. Select the IP core in the IP Catalog.
2. Click **Next** to open the Parameter Editor.
3. Click **Set Up Simulation**, which sets up all the EDA options.
4. Turn on the **Generate netlist** option to generate a netlist for resource and timing estimation and click **OK**.
5. Click **Generate** to generate the netlist file.

The Quartus II software generates a file <*output file*>**_syn.v**. This netlist contains the grey box information for resource and timing estimation, but does not contain the actual implementation. Include this netlist file in your Synplify project. Next, include the IP core variation wrapper file <*output file*>**.v|vhd** in the Quartus II project along with your Synplify **.vqm** output netlist.

If your IP core does not include a resource and timing estimation netlist, the Synplify software must treat the IP core as a black box.

Related Information

Including Files for Quartus II Placement and Routing Only on page 17-19

Instantiating Black Box IP Cores with Generated Verilog HDL Files

Use the `syn_black_box` compiler directive to declare a module as a black box. The top-level design files must contain the IP port-mapping and a hollow-body module declaration. Apply the `syn_black_box` directive to the module declaration in the top-level file or a separate file included in the project so that the Synplify software recognizes the module is a black box. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives.

The example shows a top-level file that instantiates **my_verilogIP.v**, which is a simple customized variation generated by the IP Catalog.

Example 17-6: Sample Top-Level Verilog HDL Code with Black Box Instantiation of IP

```
module top (clk, count);
    input clk;
    output [7:0] count;
    my_verilogIP verilogIP_inst (.clock (clk), .q (count));
endmodule
// Module declaration
// The following attribute is added to create a
// black box for this module.
module my_verilogIP (clock, q) /* synthesis syn_black_box */;
    input clock;
    output [7:0] q;
endmodule
```

Instantiating Black Box IP Cores with Generated VHDL Files

Use the `syn_black_box` compiler directive to declare a component as a black box. The top-level design files must contain the IP core variation component declaration and port-mapping. Apply the syn_black_box directive to the component declaration in the top-level file. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives.

The example shows a top-level file that instantiates **my_vhdlIP.vhd**, which is a simplified customized variation generated by the IP Catalog.

Example 17-7: Sample Top-Level VHDL Code with Black Box Instantiation of IP

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

ENTITY top IS
    PORT (
        clk: IN STD_LOGIC ;
        count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
END top;

ARCHITECTURE rtl OF top IS
COMPONENT my_vhdlIP
    PORT (
        clock: IN STD_LOGIC ;
        q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
end COMPONENT;
attribute syn_black_box : boolean;
attribute syn_black_box of my_vhdlIP: component is true;
BEGIN
    vhdlIP_inst : my_vhdlIP PORT MAP (
        clock => clk,
        q => count
    );
END rtl;
```

Other Synplify Software Attributes for Creating Black Boxes

Instantiating IP as a black box does not provide visibility into the IP for the synthesis tool. Thus, it does not take full advantage of the synthesis tool's timing-driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes by adding the syn_tpd, syn_tsu, and syn_tco attributes.

Example 17-8: Adding Timing Models to Black Boxes in Verilog HDL

```
module ram32x4(z,d,addr,we,clk);
    /* synthesis syn_black_box syn_tcol="clk->z[3:0]=4.0"
        syn_tpd1="addr[3:0]->[3:0]=8.0"
        syn_tsu1="addr[3:0]->clk=2.0"
        syn_tsu2="we->clk=3.0" */
    output [3:0]z;
    input[3:0]d;
    input[3:0]addr;
    input we
    input clk
endmodule
```

The following additional attributes are supported by the Synplify software to communicate details about the characteristics of the black box module within the HDL code:

- `syn_resources`—Specifies the resources used in a particular black box.
- `black_box_pad_pin`—Prevents mapping to I/O cells.
- `black_box_tri_pin`—Indicates a tri-stated signal.

For more information about applying these attributes, refer to the *Synopsys FPGA Synthesis Reference Manual*.

Including Files for Quartus II Placement and Routing Only

In the Synplify software, you can add files to your project that are used only during placement and routing in the Quartus II software. This can be useful if you have grey or black boxes for Synplify synthesis that require the full design files to be compiled in the Quartus II software.

You can also set the option in a script using the `-job_owner par` option.

The example shows how to define files for a Synplify project that includes a top-level design file, a grey box netlist file, an IP wrapper file, and an encrypted IP file. With these files, the Synplify software writes an empty instantiation of "core" in the **.vqm** file and uses the grey box netlist for resource and timing estimation. The files **core.v** and **core_enc8b10b.v** are not compiled by the Synplify software, but are copied into the place-and-route directory. The Quartus II software compiles these files to implement the "core" IP block.

Example 17-9: Commands to Define Files for a Synplify Project

```
add_file -verilog -job_owner par "core_enc8b10b.v"
add_file -verilog -job_owner par "core.v"
add_file -verilog "core_gb.v"
add_file -verilog "top.v"
```

Inferring Altera IP Cores from HDL Code

The Synplify software uses Behavior Extraction Synthesis Technology (BEST) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, and DSP multiplication operations. Then, the Synplify software keeps the structures abstract for as long as possible in the synthesis process. This allows the use of technology-specific resources to implement these structures by inferring the appropriate Altera IP core when an IP core provides optimal results.

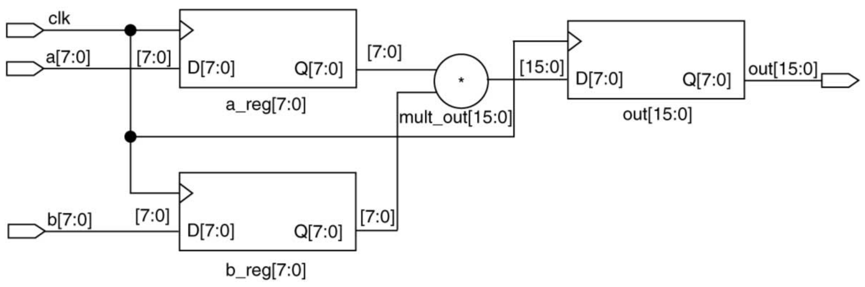
Related Information

- **Recommended HDL Coding Styles Documentation** on page 12-1

Inferring Multipliers

The figure shows the HDL Analyst view of an unsigned 8 × 8 multiplier with two pipeline stages after synthesis in the Synplify software. This multiplier is converted into an ALTMULT_ADD or ALTMULT_ACCUM IP core. For devices with DSP blocks, the software might implement the function in a DSP block instead of regular logic, depending on device utilization. For some devices, the software maps directly to DSP block device primitives instead of instantiating an IP core in the **.vqm** file.

Figure 17-2: HDL Analyst View of LPM_MULT IP Core (Unsigned 8x8 Multiplier with Pipeline=2)



Resource Balancing

While mapping multipliers to DSP blocks, the Synplify software performs resource balancing for optimum performance.

Altera devices have a fixed number of DSP blocks, which includes a fixed number of embedded multipliers. If the design uses more multipliers than are available, the Synplify software automatically maps the extra multipliers to logic elements (LEs), or adaptive logic modules (ALMs).

If a design uses more multipliers than are available in the DSP blocks, the Synplify software maps the multipliers in the critical paths to DSP blocks. Next, any wide multipliers, which might or might not be in the critical paths, are mapped to DSP blocks. Smaller multipliers and multipliers that are not in the critical paths might then be implemented in the logic (LEs or ALMs). This ensures that the design fits successfully in the device.

Controlling the DSP Block Inference

You can implement multipliers in DSP blocks or in logic in Altera devices that contain DSP blocks. You can control this implementation through attribute settings in the Synplify software.

Signal Level Attribute

You can control the implementation of individual multipliers by using the `syn_multstyle` attribute as shown in the following Verilog HDL code (where <*signal_name*> is the name of the signal):

```
<signal_name> /* synthesis syn_multstyle = "logic" */;
```

The `syn_multstyle` attribute applies to wires only; it cannot be applied to registers.

Table 17-3: DSP Block Attribute Setting in the Synplify Software

Attribute Name	Value	Description
`syn_multstyle`	`lpm_mult`	LPM function inferred and multipliers implemented in DSP blocks.
	`logic`	LPM function not inferred and multipliers implemented as LEs by the Synplify software.
	`block_mult`	DSP IP core is inferred and multipliers are mapped directly to DSP block device primitives (for supported devices).

Example 17-10: Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module mult(a,b,c,r,en);
    input [7:0] a,b;
    output [15:0] r;
    input [15:0] c;
    input en;
    wire [15:0] temp /* synthesis syn_multstyle="logic" */;

    assign temp = a*b;
    assign r = en ? temp : c;
endmodule
```

Example 17-11: Signal Attributes for Controlling DSP Block Inference in VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity onereg is port (
    r : out std_logic_vector (15 downto 0);
    en : in std_logic;
    a : in std_logic_vector (7 downto 0);
    b : in std_logic_vector (7 downto 0);
    c : in std_logic_vector (15 downto 0);
    );
end onereg;

architecture beh of onereg is
signal temp : std_logic_vector (15 downto 0);
attribute syn_multstyle : string;
attribute syn_multstyle of temp : signal is "logic";

begin
    temp <= a * b;
    r <= temp when en='1' else c;
end beh;
```

Inferring RAM

When a RAM block is inferred from an HDL design, the Synplify software uses an Altera IP core to target the device memory architecture. For some devices, the Synplify software maps directly to memory block device primitives instead of instantiating an IP core in the **.vqm** file.

Follow these guidelines for the Synplify software to successfully infer RAM in a design:

- The address line must be at least two bits wide.
- Resets on the memory are not supported. Refer to the device family documentation for information about whether read and write ports must be synchronous.
- Some Verilog HDL statements with blocking assignments might not be mapped to RAM blocks, so avoid blocking statements when modeling RAMs in Verilog HDL.

For some device families, the `syn_ramstyle` attribute specifies the implementation to use for an inferred RAM. You can apply the `syn_ramstyle` attribute globally to a module or a RAM instance, to specify `registers` or `block_ram` values. To turn off RAM inference, set the attribute value to `registers`.

When inferring RAM for some Altera device families, the Synplify software generates additional bypass logic. This logic is generated to resolve a half-cycle read/write behavior difference between the RTL and post-synthesis simulations. The RTL simulation shows the memory being updated on the positive edge of the clock; the post-synthesis simulation shows the memory being updated on the negative edge of the clock. To eliminate bypass logic, the output of the RAM must be registered. By adding this register, the output of the RAM is seen after a full clock cycle, by which time the update has occurred, thus eliminating the need for bypass logic.

For devices with TriMatrix memory blocks, disable the creation of glue logic by setting the `syn_ramstyle` value to `no_rw_check`. Set `syn_ramstyle` to `no_rw_check` to disable the creation of glue logic in dual-port mode.

Example 17-12: VHDL Code for Inferred Dual-Port RAM

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

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in: IN STD_LOGIC_VECTOR (7 DOWNTO 0)
    wr_addr, rd_addr: IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we: IN STD_LOGIC);
    clk: IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECOR (7 DOWNTO 0);
SIGNAL mem; Mem_Type;
SIGNAL addr_reg: STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN
    data_out <= mem (CONV_INTEGER(rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
                mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
        END IF;
    END PROCESS;
END ram_infer;
```

Example 17-13: VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic

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

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    wr_addr, rd_addr : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we : IN STD_LOGIC;
    clk : IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem : Mem_Type;
SIGNAL addr_reg : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL tmp_out : STD_LOGIC_VECTOR (7 DOWNTO 0); --output register

BEGIN
    tmp_out <= mem (CONV_INTEGER (rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
                mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
            data_out <= tmp_out; --registers output preventing
                                 -- bypass logic generation
        END IF;
    END PROCESS;
END ram_infer;
```

RAM Initialization

Use the Verilog HDL `$readmemb` or `$readmemh` system tasks in your HDL code to initialize RAM memories. The Synplify compiler forward-annotates the initialization values in the **.srs** (technology-independent RTL netlist) file and the mapper generates the corresponding hexadecimal memory initialization (**.hex**) file. One **.hex** file is created for each of the `altsyncram` IP cores that are inferred in the design. The **.hex** file is associated with the `altsyncram` instance in the **.vqm** file using the `init_file` attribute.

The examples show how RAM can be initialized through HDL code, and how the corresponding **.hex** file is generated using Verilog HDL.

Example 17-14: Using $readmemb System Task to Initialize an Inferred RAM in Verilog HDL Code

```
initial
begin
    $readmemb("mem.ini", mem);
end
always @(posedge clk)
begin
    raddr_reg <= raddr;
    if(we)
```

```
        mem[waddr] <= data;
end
```

Example 17-15: Sample of .vqm Instance Containing Memory Initialization File

```
altsyncram mem_hex( .wren_a(we),.wren_b(GND),...);

defparam mem_hex.lpm_type = "altsyncram";
defparam mem_hex.operation_mode = "Dual_Port";
...
defparam mem_hex.init_file = "mem_hex.hex";
```

Inferring ROM

When a ROM block is inferred from an HDL design, the Synplify software uses an Altera IP core to target the device memory architecture. For some devices, the Synplify software maps directly to memory block device atoms instead of instantiating an IP core in the **.vqm** file.

Follow these guidelines for the Synplify software to successfully infer ROM in a design:

- The address line must be at least two bits wide.
- The ROM must be at least half full.
- A CASE or IF statement must make 16 or more assignments using constant values of the same width.

Inferring Shift Registers

The Synplify software infers shift registers for sequential shift components so that they can be placed in dedicated memory blocks in supported device architectures using the ALTSHIFT_TAPS IP core.

If necessary, set the implementation style with the `syn_srlstyle` attribute. If you do not want the components automatically mapped to shift registers, set the value to `registers`. You can set the value globally, or on individual modules or registers.

For some designs, turning off shift register inference improves the design performance.

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations are made dramatically faster by focusing new compilations on particular design partitions and merging results with previous compilation results of other partitions. You can perform optimization on individual subblocks and then preserve the results before you integrate the blocks into a final design and optimize it at the top-level.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro and Premier software, provides an automated block-based incremental synthesis flow. The MultiPoint feature manages a design hierarchy to let you design incrementally and synthesize designs that take too long for synthesis of the entire project. MultiPoint synthesis allows different netlist files to be created for different sections of a design hierarchy and supports the Quartus II incremental compilation methodology. This feature also ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections.

You can also partition your design and create different netlist files manually with the Synplify software by creating a separate project for the logic in each partition of the design. Creating different netlist files for each partition of the design also means that each partition can be independent of the others.

Hierarchical design methodologies can improve the efficiency of your design process, providing better design reuse opportunities and fewer integration problems when working in a team environment. When you use these incremental synthesis methodologies, you can take advantage of incremental compilation in the Quartus II software. You can perform placement and routing on only the changed partitions of the design, which reduces place-and-route time and preserves your fitting results.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design Documentation** on page 3-1

Design Flow for Incremental Compilation

The following steps describe the general incremental compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files.
2. Determine which hierarchical blocks you want to treat as separate partitions in your design.
3. Set up your design using the MultiPoint synthesis feature or separate projects so that a separate netlist file is created for each design partition.
4. If using separate projects, disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and map each partition in the Synplify software, making constraints as you would in a non-incremental design flow.
6. Import the **.vqm** netlist and **.tcl** file for each partition into the Quartus II software and set up the Quartus II project(s) for incremental compilation.
7. Compile your design in the Quartus II software and preserve the compilation results with the post-fit netlist in incremental compilation.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the partition you modified to generate a new netlist and **.tcl** file. Do not regenerate netlist files for the unmodified partitions.
9. Import the new netlist and **.tcl** file into the Quartus II software and recompile the design in the Quartus II software with incremental compilation.

Creating a Design with Separate Netlist Files for Incremental Compilation

The first stage of a hierarchical or incremental design flow is to ensure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so you can take advantage of incremental compilation in the Quartus II software. If the entire design is in one netlist file, changes in one partition might affect other partitions because of possible node name changes when you resynthesize the design.

To ensure proper functionality of the synthesis flow, create separate netlist files only for modules and entities. In addition, each module or entity requires its own design file. If two different modules are in the same design file, but are defined as being part of different partitions, incremental compilation cannot be maintained since both partitions must be recompiled when one module is changed.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous, and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Synplify software pushes, or bubbles, the tri-states through the hierarchy to the top-level to use the tri-state drivers on output pins of Altera devices. Because bubbling tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. Use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

You can generate multiple **.vqm** netlist files with the MultiPoint synthesis flow in the Synplify Pro and Premier software, or by manually creating separate Synplify projects and creating a black box for each block that you want to designate as a separate design partition.

In the MultiPoint synthesis flow in the Synplify Pro and Premier software, you create multiple **.vqm** netlist files from one easy-to-manage, top-level synthesis project. By using the manual black box method, you have multiple synthesis projects, which might be required for certain team-based or bottom-up designs where a single top-level project is not desired.

After you have created multiple **.vqm** files using one of these two methods, you must create the appropriate Quartus II projects to place-and-route the design.

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments Documentation** on page 14-1

Using MultiPoint Synthesis with Incremental Compilation

This topic describes how to generate multiple **.vqm** files using the Synplify Pro and Premier software MultiPoint synthesis flow. You must first set up your constraint file and Synplify options, then apply the appropriate Compile Point settings to write multiple **.vqm** files and create design partition assignments for incremental compilation.

Related Information

Preserving Hierarchy on page 17-12

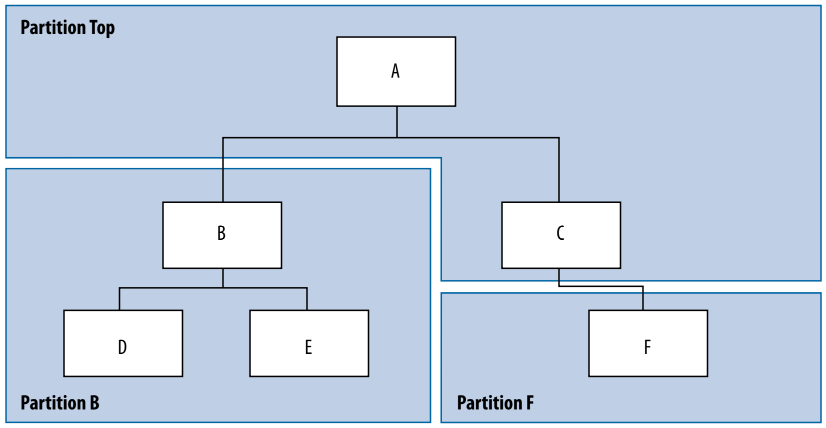
Set Compile Points and Create Constraint Files

The MultiPoint flow lets you segment a design into smaller synthesis units, called Compile Points. The synthesis software treats each Compile Point as a partition for incremental mapping, which allows you to isolate and work on each Compile Point module as independent segments of the larger design without impacting other design modules. A design can have any number of Compile Points, and Compile Points can be nested. The top-level module is always treated as a Compile Point.

Compile Points are optimized in isolation from their parent, which can be another Compile Point or a top-level design. Each block created with a Compile Point is unaffected by critical paths or constraints on its parent or other blocks. A Compile Point is independent, with its own individual constraints. During synthesis, any Compile Points that have not yet been synthesized are synthesized before the top level. Nested Compile Points are synthesized before the parent Compile Points in which they are contained. When you apply the appropriate setting for the Compile Point, a separate netlist is created for that Compile Point, isolating that logic from any other logic in the design.

The figure shows an example of a design hierarchy that is split into multiple partitions. The top-level block of each partition can be synthesized as a separate Compile Point.

Figure 17-3: Partitions in a Hierarchical Design



In this case, modules A, B, and F are Compile Points. The top-level Compile Point consists of the top-level block in the design (that is, block A in this example), including the logic that is not defined under another Compile Point. In this example, the design for top-level Compile Point A also includes the logic in one of its subblocks, C. Because block F is defined as its own Compile Point, it is not treated as part of the top-level Compile Point A. Another separate Compile Point B contains the logic in blocks B, D, and E. One netlist is created for the top-level module A and submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

Apply Compile Points to the module, or to the architecture in the Synplify Pro SCOPE spreadsheet, or to the **.sdc** file. You cannot set a Compile Point in the Verilog HDL or VHDL source code. You can set the constraints manually using Tcl, by editing the **.sdc** file, or you can use the GUI.

Defining Compile Points With .tcl or .sdc Files

To set Compile Points with a **.tcl** or **.sdc** file, use the `define_compile_point` command.

Example 17-16: The define_compile_point Command

```
define_compile_point [-disable] {<objname>} -type {locked, partition}
```

<objname> represents any module in the design. The Compile Point type `{locked, partition}` indicates that the Compile Point represents a partition for the Quartus II incremental compilation flow.

Each Compile Point has a set of constraint files that begin with the `define_current_design` command to set up the SCOPE environment, as follows:

```
define_current_design {<my_module>}
```

Additional Considerations for Compile Points

To ensure that changes to a Compile Point do not affect the top-level parent module, turn off the **Update Compile Point Timing Data** option in the **Implementation Options** dialog box. If this option is turned on, updates to a child module can impact the top-level module.

You can apply the `syn_allowed_resources` attribute to any Compile Point view to restrict the number of resources for a particular module.

When using Compile Points with incremental compilation, be aware of the following restrictions:

- To use Compile Points effectively, you must provide timing constraints (timing budgeting) for each Compile Point; the more accurate the constraints, the better your results are. Constraints are not automatically budgeted, so manual time budgeting is essential. Altera recommends that you register all inputs and outputs of each partition. This avoids any logic delay penalty on signals that cross-partition boundaries.
- When using the Synplify attribute `syn_useioff` to pack registers in the I/O Elements (IOEs) of Altera devices, these registers must be in the top-level module. Otherwise, you must direct the Quartus II software to perform I/O register packing instead of the `syn_useioff` attribute. You can use the **Fast Input Register** or **Fast Output Register** options, or set I/O timing constraints and turn on **Optimize I/O cell register placement for timing** on the **Advanced Settings (Fitter)** dialog box in the Quartus II software.
- There is no incremental synthesis support for top-level logic; any logic in the top-level is resynthesized during every compilation in the Synplify software.

For more information about using Compile Points and setting Synplify attributes and constraints for both top-level and lower-level Compile Points, refer to the *Synopsys FPGA Synthesis User Guide* and the *Synopsys FPGA Synthesis Reference Manual.*

Creating a Quartus II Project for Compile Points and Multiple .vqm Files

During compilation, the Synplify Pro and Premier software creates a <*top-level project*>**.tcl** file that provides the Quartus II software with the appropriate constraints and design partition assignments, creating a partition for each **.vqm** file along with the information to set up a Quartus II project.

Depending on your design methodology, you can create one Quartus II project for all netlists or a separate Quartus II project for each netlist. In the standard incremental compilation design flow, you create design partition assignments and optional LogicLock™ floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design.

You might require a bottom-up design flow if each partition must be optimized separately, such as for third-party IP delivery. If you use this flow, Altera recommends you create a design floorplan to avoid placement conflicts between each partition. To follow this design flow in the Quartus II software, create separate Quartus II projects, export each design partition and incorporate them into a top-level design using the incremental compilation features to maintain placement results.

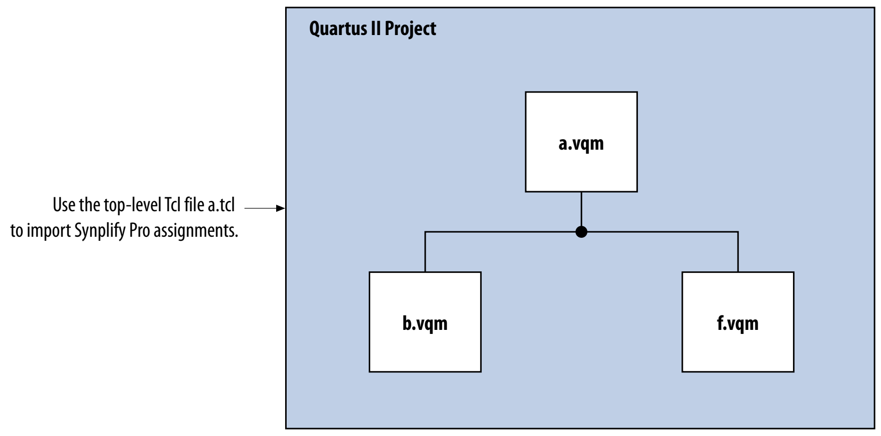
Related Information

Running the Quartus II Software Manually With the Synplify-Generated Tcl Script on page 17-7

Creating a Single Quartus II Project for a Standard Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file that contains the Synplify assignments for all partitions within the project. This method allows you to import all the partitions into one Quartus II project and optimize all modules within the project at once, while taking advantage of the performance preservation and compilation-time reduction that incremental compilation offers.

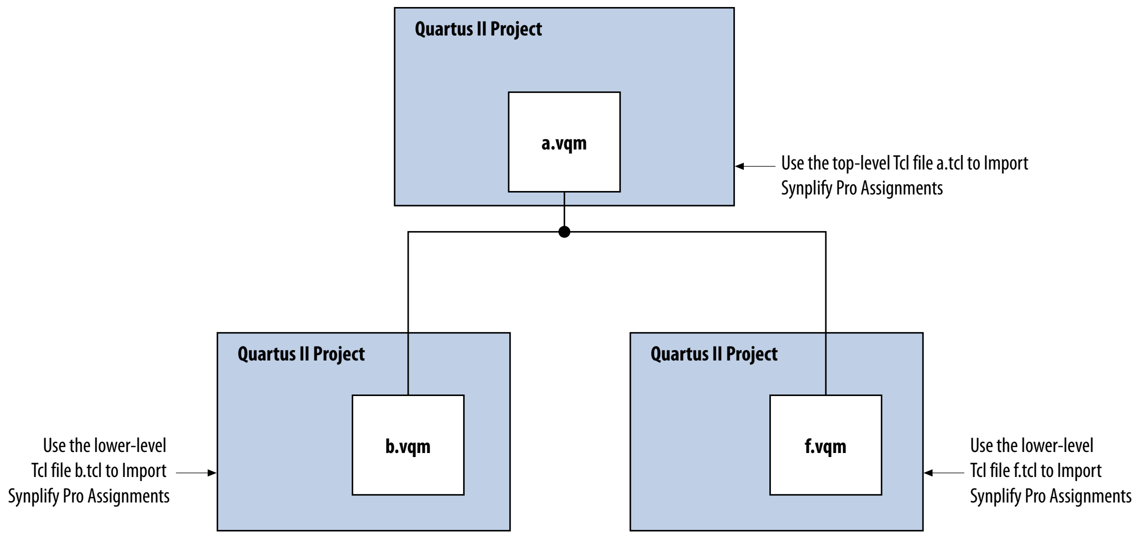
Figure 17-4: Design Flow Using Multiple .vqm Files with One Quartus II Project


Quartus II Project
a.vqm
Use the top-level Tcl file a.tcl to import Synplify Pro assignments.
b.vqm
f.vqm

Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the <*lower-level compile point*>**.tcl** files that contain the Synplify assignments for each Compile Point. Generate multiple Quartus II projects, one for each partition and netlist in the design. The designers in the project can optimize their own partitions separately within the Quartus II software and export the results for their own partitions. You can export the optimized subdesigns and then import them into one top-level Quartus II project using incremental compilation to complete the design.

Figure 17-5: Design Flow Using Multiple .vqm Files with Multiple Quartus II Projects


Quartus II Project
a.vqm
Use the top-level Tcl file a.tcl to Import Synplify Pro Assignments
Quartus II Project
b.vqm
Use the lower-level Tcl file b.tcl to Import Synplify Pro Assignments
Quartus II Project
f.vqm
Use the lower-level Tcl file f.tcl to Import Synplify Pro Assignments

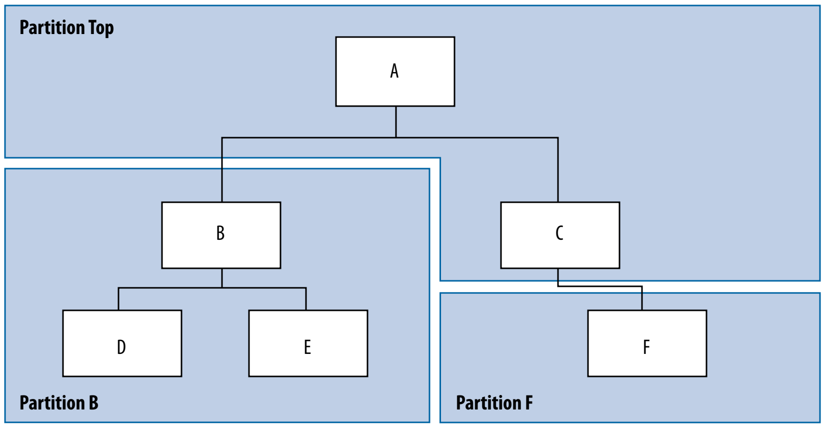
Creating Multiple .vqm Files for a Incremental Compilation Flow With Separate Synplify Projects

You can manually generate multiple **.vqm** files for a incremental compilation flow with black boxes and separate Synplify projects for each design partition. This manual flow is supported by versions of the Synplify software without the MultiPoint Synthesis feature.

Manually Creating Multiple .vqm Files With Black Boxes

To create multiple **.vqm** files manually in the Synplify software, create a separate project for each lower-level module and top-level design that you want to maintain as a separate **.vqm** file for an incremental compilation partition. Implement black box instantiations of lower-level partitions in your top-level project.

Figure 17-6: Partitions in a Hierarchical Design



The partition top contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in one of its sub-blocks, block C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, partition B, contains the logic in blocks B, D, and E. In a team-based design, engineers can work independently on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for module B and its submodules D and E, while a third netlist is created for module F.

Creating Multiple .vqm Files for this Design

To create multiple **.vqm** files for this design, follow these steps:

1. Generate a **.vqm** file for module B. Use **B.v/.vhd**, **D.v/.vhd**, and **E.v/.vhd** as the source files.
2. Generate a **.vqm** file for module F. Use **F.v/.vhd** as the source files.
3. Generate a top-level **.vqm** file for module A. Use **A.v/.vhd** and **C.v/.vhd** as the source files. Ensure that you use black box modules B and F, which were optimized separately in the previous steps.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to create a black box for the module. In Verilog HDL, you must provide an empty module declaration for a module that is treated as a black box.

The example shows the **A.v** top-level file. Follow the same procedure for lower-level files that also contain a black box for any module beneath the current level hierarchy.

Example 17-17: Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

    // Any other code in A.v goes here.
endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black boxes.

module B (data_in, clk, ld, data_out) /* synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0} data_out;
endmodule

module F (d, clk, e, q) /* synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to treat the component as a black box. In VHDL, you must have a component declaration for the black box.

Although VHDL is not case-sensitive, a **.vqm** (a subset of Verilog HDL) file is case-sensitive. Entity names and their port declarations are forwarded to the **.vqm** file. Black box names and port declarations are also passed to the **.vqm** file. To prevent case-based mismatches, use the same capitalization for black box and entity declarations in VHDL designs.

The example shows the **A.vhd** top-level file. Follow this same procedure for any lower-level files that contain a black box for any block beneath the current level of hierarchy.

Example 17-18: VHDL Black Box for Top-Level File A.vhd

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

ENTITY A IS
PORT (data_in : IN INTEGER RANGE 0 TO 15;
    clk, e, ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15 );
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk, ld : IN STD_LOGIC;
    d_out : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;

COMPONENT F PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk, e: IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15);
```

```
END COMPONENT;

attribute syn_black_box of B: component is true;
atrribute syn_black_box of F: component is true;

-- Other component declarations in A.vhd go here
signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN

U1 : B
PORT MAP (
    data_in => data_in,
    clk => clk,
    ld => ld,
    d_out => cnt_out );

U2 : F
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => data_out );

-- Any other code in A.vhd goes here

END a_arch;
```

After you complete the steps above, you have a netlist for each partition of the design. These files are ready for use with the incremental compilation flow in the Quartus II software.

Creating a Quartus II Project for Multiple .vqm Files

The Synplify software creates a **.tcl** file for each **.vqm** file that provides the Quartus II software with the appropriate constraints and information to set up a project.

Depending on your design methodology, you can create one Quartus II project for all netlists or a separate Quartus II project for each netlist. In the standard incremental compilation design flow, you create design partition assignments and optional LogicLock floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design. You might require a bottom-up design flow where each partition must be optimized separately, such as for third-party IP delivery.

To perform this design flow in the Quartus II software, create separate Quartus II projects, export each design partition and incorporate it into a top-level design using the incremental compilation features to maintain the results.

Related Information

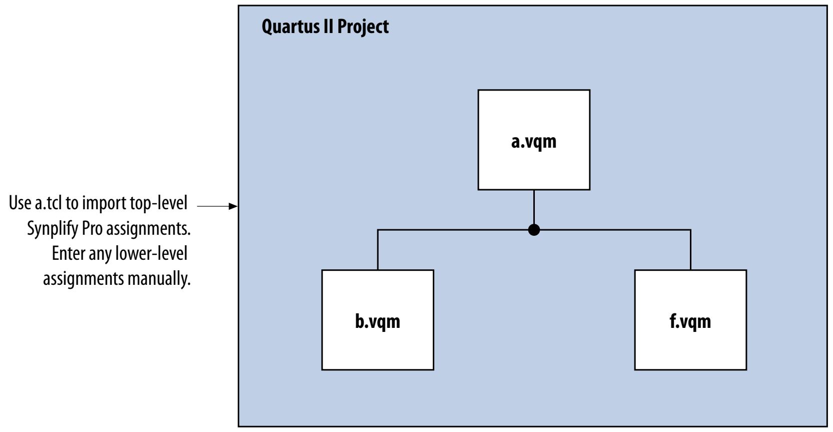
Running the Quartus II Software Manually With the Synplify-Generated Tcl Script on page 17-7

Creating a Single Quartus II Project for a Standard Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file that contains the Synplify assignments for the top-level design. This method allows you to import all of the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation time reduction offered by incremental compilation.

All of the constraints from the top-level project are passed to the Quartus II software in the top-level **.tcl** file, but constraints made in the lower-level projects within the Synplify software are not forward-annotated. Enter these constraints manually in your Quartus II project.

Figure 17-7: Design Flow Using Multiple .vqm Files with One Quartus II Project

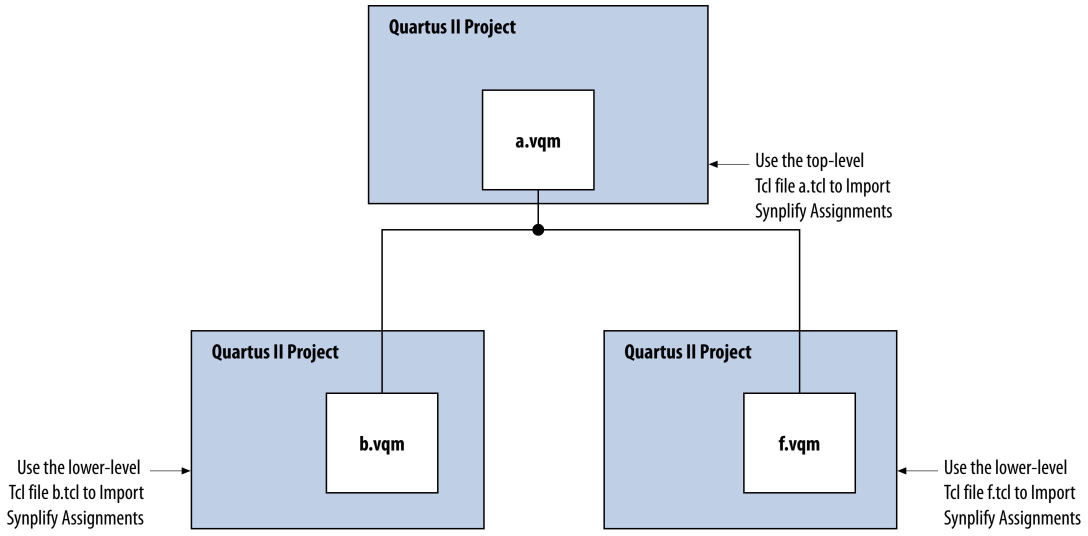

Quartus II Project
a.vqm
b.vqm
f.vqm
Use a.tcl to import top-level Synplify Pro assignments. Enter any lower-level assignments manually.

Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the **.tcl** file that is created for each **.vqm** file by the Synplify software for each Synplify project. This method generates multiple Quartus II projects, one for each block in the design. The designers in the project can optimize their own blocks separately within the Quartus II software and export the placement of their own blocks.

Designers should create a LogicLock region to create a design floorplan for each block to avoid conflicts between partitions. The top-level designer then imports all the blocks and assignments into the top-level project. This method allows each block in the design to be optimized separately and then imported into one top-level project.

Figure 17-8: Design Flow Using Multiple Synplify Projects and Multiple Quartus II Projects


Quartus II Project
a.vqm
Use the top-level Tcl file a.tcl to Import Synplify Assignments
Quartus II Project
b.vqm
Use the lower-level Tcl file b.tcl to Import Synplify Assignments
Quartus II Project
f.vqm
Use the lower-level Tcl file f.tcl to Import Synplify Assignments

Performing Incremental Compilation in the Quartus II Software

In a standard design flow using Multipoint Synthesis, the Synplify software uses the Quartus II top-level **.tcl** file to ensure that the two tools databases stay synchronized. The Tcl file creates, changes, or deletes partition assignments in the Quartus II software for Compile Points that you create, change, or delete in the Synplify software. However, if you create, change, or delete a partition in the Quartus II software, the Synplify software does not change your Compile Point settings. Make any corresponding change in your Synplify project to ensure that you create the correct **.vqm** files.

Note: If you use the NativeLink integration feature, the Synplify software does not use any information about design partition assignments that you have set in the Quartus II software.

If you create netlist files with multiple Synplify projects, or if you do not use the Synplify Pro or Premier-generated **.tcl** files to update constraints in your Quartus II project, you must ensure that your Synplify **.vqm** netlists align with your Quartus II partition settings.

After you have set up your Quartus II project with **.vqm** netlist files as separate design partitions, set the appropriate Quartus II options to preserve your compilation results. On the Assignments menu, click **Design Partitions Window**. Change the **Netlist Type** to **Post-Fit** to preserve the previous compilation's post-fit placement results. If you do not make these settings, the Quartus II software does not reuse the placement or routing results from the previous compilation.

You can take advantage of incremental compilation with your Synplify design to reduce compilation time in the Quartus II software and preserve the results for unchanged design blocks.

Related Information

Using the Quartus II Software to Run the Synplify Software on page 17-5

Quartus II Incremental Compilation for Hierarchical and Team-Based Design Documentation on page 3-1

Document Revision History

Table 17-4: Document Revision History

Date	Version	Changes
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.
November 2013	13.1.0	Dita conversion. Restructured content.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.

Date	Version	Changes
December 2010	10.1.0	• Changed to new document template. • Removed Classic Timing Analyzer support. • Removed the "altera_implement_in_esb or altera_implement_in_eab" section. • Edited the "Creating a Quartus II Project for Compile Points and Multiple .vqm Files" on page 14–33 section for changes with the incremental compilation flow. • Edited the "Creating a Quartus II Project for Multiple .vqm Files" on page 14–39 section for changes with the incremental compilation flow. • Editorial changes.
July 2010	10.0.0	• Minor updates for the Quartus II software version 10.0 release.
November 2009	9.1.0	• Minor updates for the Quartus II software version 9.1 release.
March 2009	9.0.0	• Added new section "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 14–14. • Minor updates for the Quartus II software version 9.0 release. • Chapter 10 was previously Chapter 9 in software version 8.1.

Date	Version	Changes
November 2008	8.1.0	• Changed to 8-1/2 x 11 page size • Changed the chapter title from “Synplicity Synplify & Synplify Pro Support” to “Synopsys Synplify Support” • Replaced references to Synplicity with references to Synopsys • Added information about Synplify Premier • Updated supported device list • Added SystemVerilog information to Figure 14–1

Date	Version	Changes
May 2008	8.0.0	• Updated supported device list • Updated constraint annotation information for the TimeQuest Timing Analyzer • Updated RAM and MAC constraint limitations • Revised Table 9–1 • Added new section “Changing Synplify’s Default Behavior for Instantiated Altera Megafunctions” • Added new section “Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench” • Added new section “Including Files for Quartus II Placement and Routing Only” • Added new section “Additional Considerations for Compile Points” • Removed section “Apply the LogicLock Attributes” • Modified Figure 9–4, 9–43, 9–47. and 9–48 • Added new section “Performing Incremental Compilation in the Quartus II Software” • Numerous text changes and additions throughout the chapter • Renamed several sections • Updated “Referenced Documents” section

Related Information

Quartus II Handbook Archive Website

Mentor Graphics Precision Synthesis Support 18

2014.06.30

QII5V1 Subscribe Send Feedback

About Precision RTL Synthesis Support

This manual delineates the support for the Mentor Graphics® Precision RTL Synthesis and Precision RTL Plus Synthesis software in the Quartus® II software, as well as key design flows, methodologies and techniques for improving your results for Altera® devices. This manual assumes that you have set up, licensed, and installed the Precision Synthesis software and the Quartus II software. You must set up, license, and install the Precision RTL Plus Synthesis software if you want to use the incremental synthesis feature for incremental compilation and block-based design.

To obtain and license the Precision Synthesis software, refer to the Mentor Graphics website. To install and run the Precision Synthesis software and to set up your work environment, refer to the *Precision Synthesis Installation Guide* in the Precision Manuals Bookcase. To access the Manuals Bookcase in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Related Information

Mentor Graphics website

Design Flow

The following steps describe a basic Quartus II design flow using the Precision Synthesis software:

1. Create Verilog HDL or VHDL design files.
2. Create a project in the Precision Synthesis software that contains the HDL files for your design, select your target device, and set global constraints.
3. Compile the project in the Precision Synthesis software.
4. Add specific timing constraints, optimization attributes, and compiler directives to optimize the design during synthesis. With the design analysis and cross-probing capabilities of the Precision Synthesis software, you can identify and improve circuit area and performance issues using prelayout timing estimates.

   Note: For best results, Mentor Graphics recommends specifying constraints that are as close as possible to actual operating requirements. Properly setting clock and I/O constraints, assigning clock domains, and indicating false and multicycle paths guide the synthesis algorithms more accurately toward a suitable solution in the shortest synthesis time.
5. Synthesize the project in the Precision Synthesis software.
6. Create a Quartus II project and import the following files generated by the Precision Synthesis software into the Quartus II project:

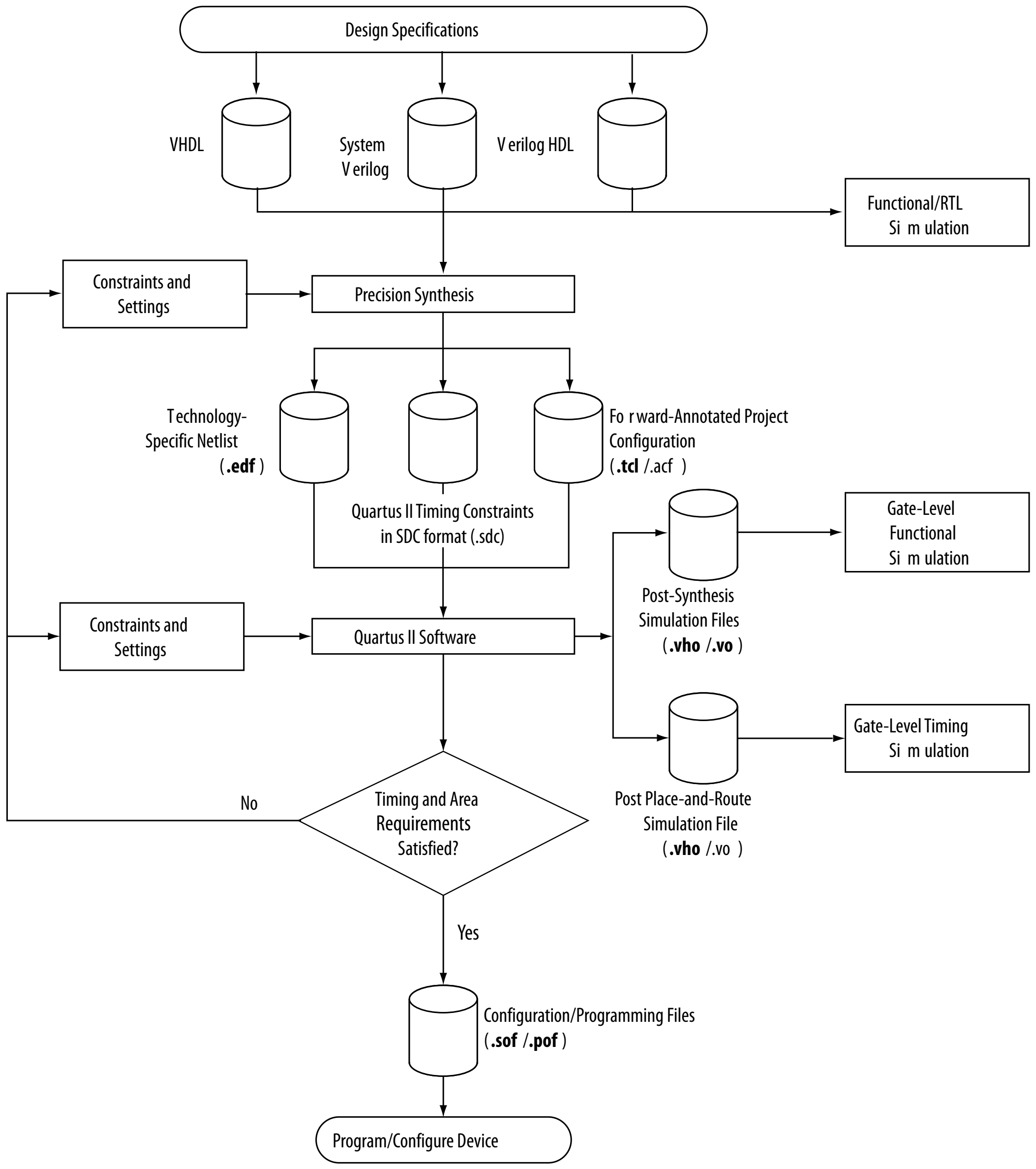
- The Verilog Quartus Mapping File (**.vqm**) netlist
- Synopsys Design Constraints File (**.sdc**) for TimeQuest Timing Analyzer constraints
- Tcl Script Files (**.tcl**) to set up your Quartus II project and pass constraints

Note: If your design uses the Classic Timing Analyzer for timing analysis in the Quartus II software versions 10.0 and earlier, the Precision Synthesis software generates timing constraints in the Tcl Constraints File (**.tcl**). If you are using the Quartus II software versions 10.1 and later, you must use the TimeQuest Timing Analyzer for timing analysis.

7. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

You can run the Quartus II software from within the Precision Synthesis software, or run the Precision Synthesis software using the Quartus II software.

Figure 18-1: Design Flow Using the Precision Synthesis Software and Quartus II Software


Design Specifications
VHDL
System Verilog
Verilog HDL
Functional/RTL Simulation
Constraints and Settings
Precision Synthesis
Technology-Specific Netlist (.edf)
Forward-Annotated Project Configuration (.tcl/.acf)
Quartus II Timing Constraints in SDC format (.sdc)
Gate-Level Functional Simulation
Post-Synthesis Simulation Files (.vho/.vo)
Constraints and Settings
Quartus II Software
Gate-Level Timing Simulation
No
Timing and Area Requirements Satisfied?
Post Place-and-Route Simulation File (.vho/.vo)
Yes
Configuration/Programming Files (.sof/.pof)
Program/Configure Device

Related Information

- **Running the Quartus II Software from within the Precision Synthesis Software** on page 18-9
- **Using the Quartus II Software to Run the Precision Synthesis Software** on page 18-10

Timing Optimization

If your area or timing requirements are not met, you can change the constraints and resynthesize the design in the Precision Synthesis software, or you can change the constraints to optimize the design during place-and-route in the Quartus II software. Repeat the process until the area and timing requirements are met.

You can use other options and techniques in the Quartus II software to meet area and timing requirements. For example, the **WYSIWYG Primitive Resynthesis** option can perform optimizations on your EDIF netlist in the Quartus II software.

While simulation and analysis can be performed at various points in the design process, final timing analysis should be performed after placement and routing is complete.

Related Information

- Netlist Optimizations and Physical Synthesis documentation
- Timing Closure and Optimization documentation

Altera Device Family Support

The Precision Synthesis software supports active devices available in the current version of the Quartus II software. Support for newly released device families may require an overlay. Contact Mentor Graphics for more information.

Precision Synthesis Generated Files

During synthesis, the Precision Synthesis software produces several intermediate and output files.

Table 18-1: Precision Synthesis Software Intermediate and Output Files

File Extension	File Description
.psp	Precision Synthesis Project File.
.xdb	Mentor Graphics Design Database File.
.rep(15)	Synthesis Area and Timing Report File.
.vqm(16)	Technology-specific netlist in **.vqm** file format. By default, the Precision Synthesis software creates **.vqm** files for Arria series, Cyclone series, and Stratix series devices. The Precision Synthesis software defaults to creating **.vqm** files when the device is supported.

(15) The timing report file includes performance estimates that are based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route for accurate post-place-and-route timing information. The area report file includes post-synthesis device resource utilization statistics that can differ from the resource usage after place-and-route due to black boxes or further optimizations performed during placement and routing. Use the device utilization reported by the Quartus II software after place-and-route for final resource utilization results.

(16) The Precision Synthesis software-generated VQM file is supported by the Quartus II software version 10.1 and later.

File Extension	File Description
.tcl	Forward-annotated Tcl assignments and constraints file. The *<project name>*.**tcl** file is generated for all devices. The **.tcl** file acts as the Quartus II Project Configuration file and is used to make basic project and placement assignments, and to create and compile a Quartus II project.
.acf	Assignment and Configurations file for backward compatibility with the MAX +PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II **.acf** file.
.sdc	Quartus II timing constraints file in Synopsys Design Constraints format. This file is generated automatically if the device uses the TimeQuest Timing Analyzer by default in the Quartus II software, and has the naming convention *<project name>*_**pnr_constraints** **.sdc**.

Related Information

- **Exporting Designs to the Quartus II Software Using NativeLink Integration** on page 18-9
- **Synthesizing the Design and Evaluating the Results** on page 18-8

Creating and Compiling a Project in the Precision Synthesis Software

After creating your design files, create a project in the Precision Synthesis software that contains the basic settings for compiling the design.

Mapping the Precision Synthesis Design

In the next steps, you set constraints and map the design to technology-specific cells. The Precision Synthesis software maps the design by default to the fastest possible implementation that meets your timing constraints. To accomplish this, you must specify timing requirements for the automatically determined clock sources. With this information, the Precision Synthesis software performs static timing analysis to determine the location of the critical timing paths. The Precision Synthesis software achieves the best results for your design when you set as many realistic constraints as possible. Be sure to set constraints for timing, mapping, false paths, multicycle paths, and other factors that control the structure of the implemented design.

Mentor Graphics recommends creating an **.sdc** file and adding this file to the **Constraint Files** section of the **Project Files** list. You can create this file with a text editor, by issuing command-line constraint parameters, or by directing the Precision Synthesis software to generate the file automatically the first time you synthesize your design. By default, the Precision Synthesis software saves all timing constraints and attributes in two files: **precision_rtl.sdc** and **precision_tech.sdc**. The **precision_rtl.sdc** file contains constraints set on the RTL-level database (post-compilation) and the **precision_tech.sdc** file contains constraints set on the gate-level database (post- synthesis) located in the current implementation directory.

You can also enter constraints at the command line. After adding constraints at the command line, update the **.sdc** file with the `update constraint file` command. You can add constraints that change infrequently directly to the HDL source files with HDL attributes or pragmas.

Note: The Precision **.sdc** file contains all the constraints for the Precision Synthesis project. For the Quartus II software, placement constraints are written in a **.tcl** file and timing constraints for the TimeQuest Timing Analyzer are written in the Quartus II **.sdc** file.

For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis User's Manual* and the *Precision Synthesis Reference Manual*. For more details and examples of attributes, refer to the *Attributes* chapter in the *Precision Synthesis Reference Manual.*

Setting Timing Constraints

The Precision Synthesis software uses timing constraints, based on the industry-standard **.sdc** file format, to deliver optimal results. Missing timing constraints can result in incomplete timing analysis and might prevent timing errors from being detected. The Precision Synthesis software provides constraint analysis prior to synthesis to ensure that designs are fully and accurately constrained. The <*project name*>**_pnr_constraints.sdc** file, which contains timing constraints in SDC format, is generated in the Quartus II software.

Note: Because the **.sdc** file format requires that timing constraints be set relative to defined clocks, you must specify your clock constraints before applying any other timing constraints.

You also can use multicycle path and false path assignments to relax requirements or exclude nodes from timing requirements, which can improve area utilization and allow the software optimizations to focus on the most critical parts of the design.

For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis User's Manual* and the *Precision Synthesis Reference Manual*.

Setting Mapping Constraints

Mapping constraints affect how your design is mapped into the target Altera device. You can set mapping constraints in the user interface, in HDL code, or with the `set_attribute` command in the constraint file.

Assigning Pin Numbers and I/O Settings

The Precision Synthesis software supports assigning device pin numbers, I/O standards, drive strengths, and slew-rate settings to top-level ports of the design. You can set these timing constraints with the `set_attribute` command, the GUI, or by specifying synthesis attributes in your HDL code. These constraints are forward-annotated in the <*project name*>**.tcl** file that is read by the Quartus II software during place-and-route and do not affect synthesis.

You can use the `set_attribute` command in the Precision Synthesis software **.sdc** file to specify pin number constraints, I/O standards, drive strengths, and slow slew-rate settings. The table below describes the format to use for entries in the Precision Synthesis software constraint file.

Table 18-2: Constraint File Settings

Constraint	Entry Format for Precision Constraint File
Pin number	`set_attribute -name PIN_NUMBER -value "<pin number>" -port <port name>`
I/O standard	`set_attribute -name IOSTANDARD -value "<I/O Standard>" -port <port name>`
Drive strength	`set_attribute -name DRIVE -value "<drive strength in mA>" -port <port name>`
Slew rate	`set_attribute -name SLEW -value "TRUE | FALSE" -port <port name>`

You also can use synthesis attributes or pragmas in your HDL code to make these assignments.

Example 18-1: Verilog HDL Pin Assignment

```
//pragma attribute clk pin_number P10;
```

Example 18-2: VHDL Pin Assignment

```
attribute pin_number : string
attribute pin_number of clk : signal is "P10";
```

You can use the same syntax to assign the I/O standard using the `IOSTANDARD` attribute, drive strength using the attribute `DRIVE`, and slew rate using the `SLEW` attribute.

For more details about attributes and how to set these attributes in your HDL code, refer to the *Precision Synthesis Reference Manual.*

Assigning I/O Registers

The Precision Synthesis software performs timing-driven I/O register mapping by default. You can force a register to the device IO element (IOE) using the Complex I/O constraint. This option does not apply if you turn off **I/O pad insertion.**

Note: You also can make the assignment by right-clicking on the pin in the Schematic Viewer.

For the Stratix series, Cyclone series, and the MAX II device families, the Precision Synthesis software can move an internal register to an I/O register without any restrictions on design hierarchy.

For more mature devices, the Precision Synthesis software can move an internal register to an I/O register only when the register exists in the top-level of the hierarchy. If the register is buried in the hierarchy, you must flatten the hierarchy so that the buried registers are moved to the top-level of the design.

Disabling I/O Pad Insertion

The Precision Synthesis software assigns I/O pad atoms (device primitives used to represent the I/O pins and I/O registers) to all ports in the top-level of a design by default. In certain situations, you might not want the software to add I/O pads to all I/O pins in the design. The Quartus II software can compile a design without I/O pads; however, including I/O pads provides the Precision Synthesis software with more information about the top-level pins in the design.

Preventing the Precision Synthesis Software from Adding I/O Pads

If you are compiling a subdesign as a separate project, I/O pins cannot be primary inputs or outputs of the device; therefore, the I/O pins should not have an I/O pad associated with them.

To prevent the Precision Synthesis software from adding I/O pads:

- You can use the Precision Synthesis GUI or add the following command to the project file:

```
setup_design -addio=false
```

Preventing the Precision Synthesis Software from Adding an I/O Pad on an Individual Pin

To prevent I/O pad insertion on an individual pin when you are using a black box, such as DDR or a phase-locked loop (PLL), at the external ports of the design, perform the following steps:

1. Compile your design.
2. Use the Precision Synthesis GUI to select the individual pin and turn off I/O pad insertion.

Note: You also can make this assignment by attaching the `nopad` attribute to the port in the HDL source code.

Controlling Fan-Out on Data Nets

Fan-out is defined as the number of nodes driven by an instance or top-level port. High fan-out nets can cause significant delays that result in an unroutable net. On a critical path, high fan-out nets can cause longer delays in a single net segment that result in the timing constraints not being met. To prevent this behavior, each device family has a global fan-out value set in the Precision Synthesis software library. In addition, the Quartus II software automatically routes high fan-out signals on global routing lines in the Altera device whenever possible.

To eliminate routability and timing issues associated with high fan-out nets, the Precision Synthesis software also allows you to override the library default value on a global or individual net basis. You can override the library value by setting a `max_fanout` attribute on the net.

Synthesizing the Design and Evaluating the Results

During synthesis, the Precision Synthesis software optimizes the compiled design, and then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the following naming convention:

```
<project name>_impl_<number>
```

After synthesis is complete, you can evaluate the results for area and timing. The *Precision RTL Synthesis User's Manual* describes different results that can be evaluated in the software.

There are several schematic viewers available in the Precision Synthesis software: RTL schematic, Technology-mapped schematic, and Critical Path schematic. These analysis tools allow you to quickly and easily isolate the source of timing or area issues, and to make additional constraint or code changes to optimize the design.

Obtaining Accurate Logic Utilization and Timing Analysis Reports

Historically, designers have relied on post-synthesis logic utilization and timing reports to determine the amount of logic their design requires, the size of the device required, and how fast the design runs. However, today's FPGA devices provide a wide variety of advanced features in addition to basic registers and look-up tables (LUTs). The Quartus II software has advanced algorithms to take advantage of these features, as well as optimization techniques to increase performance and reduce the amount of logic required for a given design. In addition, designs can contain black boxes and functions that take advantage of specific device features. Because of these advances, synthesis tool reports provide post-synthesis area and timing estimates, but you should use the place-and-route software to obtain final logic utilization and timing reports.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, which allows you to run other EDA design entry/synthesis, simulation, and timing analysis tools automatically from within the Quartus II software.

After a design is synthesized in the Precision Synthesis software, the technology-mapped design is written to the current implementation directory as an EDIF netlist file, along with a Quartus II Project Configuration File and a place-and-route constraints file. You can use the Project Configuration script, <*project name*>**.tcl**, to create and compile a Quartus II project for your EDIF or VQM netlist. This script makes basic project assignments, such as assigning the target device specified in the Precision Synthesis software. If you select a newer Altera device, the constraints are written in SDC format to the <*project name*>_ **pnr_constraints.sdc** file by default, which is used by the Fitter and the TimeQuest Timing Analyzer in the Quartus II software.

Use the following Precision Synthesis software command before compilation to generate the <*project name*>**_pnr_constraints.sdc**:

```
setup_design -timequest_sdc
```

With this command, the file is generated after synthesis.

Running the Quartus II Software from within the Precision Synthesis Software

The Precision Synthesis software also has a built-in place-and-route environment that allows you to run the Quartus II Fitter and view the results in the Precision Synthesis GUI. This feature is useful when performing an initial compilation of your design to view post-place-and-route timing and device utilization results. Not all the advanced Quartus II options that control the compilation process are available when you use this feature.

Two primary Precision Synthesis software commands control the place-and-route process. Use the `setup_place_and_route` command to set the place-and-route options. Start the process with the `place_and_route` command.

Precision Synthesis software uses individual Quartus II executables, such as analysis and synthesis (`quartus_map`), Fitter (`quartus_fit`), and the TimeQuest Timing Analyzer (`quartus_sta`) for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus II Modular** flow option in the Precision Synthesis software. By default, the Precision Synthesis software generates a Quartus II Project Configuration File (**.tcl** file) for current device families. Timing constraints that you set during synthesis are exported to the Quartus II place-and-route constraints file <*project name*>**_pnr_constraints.sdc**.

After you compile the design in the Quartus II software from within the Precision Synthesis software, you can invoke the Quartus II GUI manually and then open the project using the generated Quartus II project file. You can view reports, run analysis tools, specify options, and run the various processing flows available in the Quartus II software.

For more information about running the Quartus II software from within the Precision Synthesis software, refer to the *Altera Quartus II Integration* chapter in the *Precision Synthesis Reference Manual.*

Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script

You can run the Quartus II software using a Tcl script generated by the Precision Synthesis software. To run the Tcl script generated by the Precision Synthesis software to set up your project and start a full compilation, perform the following steps:

1. Ensure the **.vqm** file, **.tcl** files, and **.sdc** file are located in the same directory. The files should be located in the implementation directory by default.
2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**.
3. At the Tcl Console command prompt, type the command:

```
source <path>/<project name>.tcl
```

4. On the File menu, click **Open Project**. Browse to the project name and click **Open**.
5. Compile the project in the Quartus II software.

Using the Quartus II Software to Run the Precision Synthesis Software

With NativeLink integration, you can set up the Quartus II software to run the Precision Synthesis software. This feature allows you to use the Precision Synthesis software to synthesize a design as part of a standard compilation. When you use this feature, the Precision Synthesis software does not use any timing constraints or assignments, such as incremental compilation partitions, that you have set in the Quartus II software.

Related Information

- **Exporting Designs to the Quartus II Software Using NativeLink Integration** on page 18-9
- **Using the NativeLink Feature with Other EDA Tools online help**

Passing Constraints to the Quartus II Software

The place-and-route constraints script forward-annotates timing constraints that you made in the Precision Synthesis software. This integration allows you to enter these constraints once in the Precision Synthesis software, and then pass them automatically to the Quartus II software.

The following constraints are translated by the Precision Synthesis software and are applicable to the TimeQuest Timing Analyzer:

- `create_clock`
- `set_input_delay`
- `set_output_delay`
- `set_max_delay`

- `set_min_delay`
- `set_false_path`
- `set_multicycle_path`

create_clock

You can specify a clock in the Precision Synthesis software.

Example 18-3: Specifying a Clock Using create_clock

```
create_clock -name <clock_name> -period <period in ns> \
-waveform {<edge_list>} -domain <ClockDomain> <pin>
```

The period is specified in units of nanoseconds (ns). If no clock domain is specified, the clock belongs to a default clock domain `main`. All clocks in the same clock domain are treated as synchronous (related) clocks. If no <*clock_name*> is provided, the default name `virtual_default` is used. The <*edge_list*> sets the rise and fall edges of the clock signal over an entire clock period. The first value in the list is a rising transition, typically the first rising transition after time zero. The waveform can contain any even number of alternating edges, and the edges listed should alternate between rising and falling. The position of any edge can be equal to or greater than zero but must be equal to or less than the clock period.

If `-waveform <edge_list>` is not specified and `-period <period in ns>` is specified, the default waveform has a rising edge of 0.0 and a falling edge of <*period_value*>/2.

The Precision Synthesis software maps the clock constraint to the TimeQuest `create_clock` setting in the Quartus II software.

The Quartus II software supports only clock waveforms with two edges in a clock cycle. If the Precision Synthesis software finds a multi-edge clock, it issues an error message when you synthesize your design in the Precision Synthesis software.

set_input_delay

This port-specific input delay constraint is specified in the Precision Synthesis software.

Example 18-4: Specifying set_input_delay

```
set_input_delay {<delay_value> <port_pin_list>} \
-clock <clock_name> -rise -fall -add_delay
```

This constraint is mapped to the `set_input_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The input pin name for the assignment can be an input pin name of a time group. The software can use the `clock_fall` option to specify delay relative to the falling edge of the clock.

Note: Although the Precision Synthesis software allows you to set input delays on pins inside the design, these constraints are not sent to the Quartus II software, and a message is displayed.

set_output_delay

This port-specific output delay constraint is specified in the Precision Synthesis software.

Example 18-5: Using the set_output_delay Constraint

```
set_output_delay {<delay_value> <port_pin_list>} \
-clock <clock_name> -rise -fall -add_delay
```

This constraint is mapped to the `set_output_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The output pin name for the assignment can be an output pin name of a time group.

Note: Although the Precision Synthesis software allows you to set output delays on pins inside the design, these constraints are not sent to the Quartus II software.

set_max_delay and set_min_delay

The maximum delay and minimum delay for a point-to-point timing path constraint is specified in the Precision Synthesis software.

Example 18-6: Using the set_max_delay Constraint

```
set_max_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

Example 18-7: Using the set_min_delay Constraint

```
set_min_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

The `set_max_delay` and `set_min_delay` commands specify that the maximum and minimum respectively, required delay for any start point in <*from_node_list*> to any endpoint in <*to_node_list*> must be less than or greater than <*delay_value*>. Typically, you use these commands to override the default setup constraint for any path with a specific maximum or minimum time value for the path.

The node lists can contain a collection of clocks, registers, ports, pins, or cells. The `-from` and `-to` parameters specify the source (start point) and the destination (endpoint) of the timing path, respectively. The source list (<*from_node_list*>) cannot include output ports, and the destination list (<*to_node_list*>) cannot include input ports. If you include more than one node on a list, you must enclose the nodes in quotes or in braces ({ }).

If you specify a clock in the source list, you must specify a clock in the destination list. Applying `set_max_delay` or `set_min_delay` setting between clocks applies the exception from all registers or ports driven by the source clock to all registers or ports driven by the destination clock. Applying exceptions between clocks is more efficient than applying them for specific node-to-node, or node-to-clock paths. If you want to specify pin names in the list, the source must be a clock pin and the destination must be any non-clock input pin to a register. Assignments from clock pins, or to and from cells, apply to all registers in the cell or for those driven by the clock pin.

set_false_path

The false path constraint is specified in the Precision Synthesis software.

Example 18-8: Using the set_false_path Constraint

```
set_false_path -to <to_node_list> -from <from_node_list> -reset_path
```

The node lists can be a list of clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as * and ?.

In a place-and-route Tcl constraints file, this false path setting in the Precision Synthesis software is mapped to a `set_false_path` setting. The Quartus II software supports `setup`, `hold`, `rise`, or `fall` options for this assignment.

The node lists for this assignment represents top-level ports and/or nets connected to instances (end points of timing assignments).

Any false path setting in the Precision Synthesis software can be mapped to a setting in the Quartus II software with a `through` path specification.

set_multicycle_path

The multicycle path constraint is specified in the Precision Synthesis software.

Example 18-9: Using the set_multicycle_path Constraint

```
set_multicycle_path <multiplier_value> [-start] [-end] \
-to <to_node_list> -from <from_node_list> -reset_path
```

The node list can contain clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as `*` and `?`. Paths without multicycle path definitions are identical to paths with multipliers of 1. To add one additional cycle to the datapath, use a multiplier value of 2. The option `start` indicates that source clock cycles should be considered for the multiplier. The option `end` indicates that destination clock cycles should be considered for the multiplier. The default is to reference the end clock.

In the place-and-route Tcl constraints file, the multicycle path setting in the Precision Synthesis software is mapped to a `set_multicycle_path` setting. The Quartus II software supports the `rise` or `fall` options on this assignment.

The node lists represent top-level ports and/or nets connected to instances (end points of timing assignments). The node lists can contain wildcards (such as `*`); the Quartus II software automatically expands all wildcards.

Any multicycle path setting in Precision Synthesis software can be mapped to a setting in the Quartus II software with a `-through` specification.

Guidelines for Altera IP Cores and Architecture-Specific Features

Altera provides parameterizable IP cores, including the LPMs, device-specific Altera IP cores, and IP available through the Altera Megafunction Partners Program (AMPP^{SM}). You can use IP cores by instantiating them in your HDL code or by inferring certain functions from generic HDL code.

If you want to instantiate an IP core such as a PLL in your HDL code, you can instantiate and parameterize the function using the port and parameter definitions, or you can customize a function with the Parameter Editor. Altera recommends using the IP Catalog and Parameter Editor, which provides a graphical interface within the Quartus II software for customizing and parameterizing any available IP core for the design.

The Precision Synthesis software automatically recognizes certain types of HDL code and infers the appropriate IP core.

Related Information

- **Inferring Altera IP Cores from HDL Code** on page 18-16
- **Recommended HDL Coding Styles documentation** on page 12-1
- **Introduction to Altera IP Cores documentation**

Instantiating IP Cores With IP Catalog-Generated Verilog HDL Files

The IP Catalog generates a Verilog HDL instantiation template file <*output file*>**_inst.v** and a hollow-body black box module declaration <*output file*>**_bb.v** for use in your Precision Synthesis design. Incorporate the instantiation template file, <*output file*>**_inst.v**, into your top-level design to instantiate the IP core wrapper file, <*output file*>**.v**.

Include the hollow-body black box module declaration <*output file*>**_bb.v** in your Precision Synthesis project to describe the port connections of the black box. Adding the IP core wrapper file <*output file*>**.v** in your Precision Synthesis project is optional, but you must add it to your Quartus II project along with the Precision Synthesis-generated EDIF or VQM netlist.

Alternatively, you can include the IP core wrapper file <*output file*>**.v** in your Precision Synthesis project and turn on the **Exclude file from Compile Phase** option in the Precision Synthesis software to exclude the file from compilation and to copy the file to the appropriate directory for use by the Quartus II software during place-and-route.

Instantiating IP Cores With IP Catalog-Generated VHDL Files

The IP Catalog generates a VHDL component declaration file <*output file*>**.cmp** and a VHDL instantiation template file <*output file*>**_inst.vhd** for use in your Precision Synthesis design. Incorporate the component declaration and instantiation template into your top-level design to instantiate the IP core wrapper file, <*output file*>**.vhd**.

Adding the IP core wrapper file <*output file*>**.vhd** in your Precision Synthesis project is optional, but you must add the file to your Quartus II project along with the Precision Synthesis-generated EDIF or VQM netlist.

Alternatively, you can include the IP core wrapper file <*output file*>**.v** in your Precision Synthesis project and turn on the **Exclude file from Compile Phase** option in the Precision Synthesis software to exclude the file from compilation and to copy the file to the appropriate directory for use by the Quartus II software during place-and-route.

Instantiating Intellectual Property With the IP Catalog and Parameter Editor

Many Altera IP functions include a resource and timing estimation netlist that the Precision Synthesis software can use to synthesize and optimize logic around the IP efficiently. As a result, the Precision Synthesis software provides better timing correlation, area estimates, and Quality of Results (QoR) than a black box approach.

To create this netlist file, perform the following steps:

1. Select the IP function in the IP Catalog.
2. Click **Next** to open the Parameter Editor.
3. Click **Set Up Simulation**, which sets up all the EDA options.
4. Turn on the **Generate netlist** option to generate a netlist for resource and timing estimation and click **OK**.
5. Click **Generate** to generate the netlist file.

The Quartus II software generates a file *<output file>***_syn.v**. This netlist contains the "grey box" information for resource and timing estimation, but does not contain the actual implementation. Include this netlist file into your Precision Synthesis project as an input file. Then include the IP core wrapper file *<output file>***.v|vhd** in the Quartus II project along with your EDIF or VQM output netlist.

The generated "grey box" netlist file, *<output file>***_syn.v** , is always in Verilog HDL format, even if you select VHDL as the output file format.

Note: For information about creating a grey box netlist file from the command line, search Altera's Knowledge Database.

Related Information

Altera Knowledge Center website

Instantiating Black Box IP Functions With Generated Verilog HDL Files

You can use the `syn_black_box` or `black_box` compiler directives to declare a module as a black box. The top-level design files must contain the IP port mapping and a hollow-body module declaration. You can apply the directive to the module declaration in the top-level file or a separate file included in the project so that the Precision Synthesis software recognizes the module is a black box.

Note: The `syn_black_box` and `black_box` directives are supported only on module or entity definitions.

The example below shows a sample top-level file that instantiates **my_verilogIP.v**, which is a simplified customized variation generated by the IP Catalog and Parameter Editor.

Example 18-10: Top-Level Verilog HDL Code with Black Box Instantiation of IP

```
module top (clk, count);
   input clk;
   output[7:0] count;

   my_verilogIP verilogIP_inst (.clock (clk), .q (count));
endmodule

// Module declaration
// The following attribute is added to create a
// black box for this module.
module my_verilogIP (clock, q) /* synthesis syn_black_box */;
   input clock;
```

```
    output[7:0] q;
endmodule
```

Instantiating Black Box IP Functions With Generated VHDL Files

You can use the `syn_black_box` or `black_box` compiler directives to declare a component as a black box. The top-level design files must contain the IP core variation component declaration and port mapping. Apply the directive to the component declaration in the top-level file.

Note: The `syn_black_box` and `black_box` directives are supported only on module or entity definitions.

The example below shows a sample top-level file that instantiates **my_vhdlIP.vhd**, which is a simplified customized variation generated by the IP Catalog and Parameter Editor.

Example 18-11: Top-Level VHDL Code with Black Box Instantiation of IP

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY top IS
  PORT (
    clk: IN STD_LOGIC ;
    count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END top;

ARCHITECTURE rtl OF top IS
  COMPONENT my_vhdlIP
  PORT (
    clock: IN STD_LOGIC ;
    q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
  end COMPONENT;
  attribute syn_black_box : boolean;
  attribute syn_black_box of my_vhdlIP: component is true;
  BEGIN
    vhdlIP_inst : my_vhdlIP PORT MAP (
       clock => clk,
       q => count
    );
END rtl;
```

Inferring Altera IP Cores from HDL Code

The Precision Synthesis software automatically recognizes certain types of HDL code and maps arithmetical operators, relational operators, and memory (RAM and ROM), to technology-specific implementations. This functionality allows technology-specific resources to implement these structures by inferring the appropriate Altera function to provide optimal results. In some cases, the Precision Synthesis software has options that you can use to disable or control inference.

For coding style recommendations and examples for inferring technology-specific architecture in Altera devices, refer to the *Precision Synthesis Style Guide*.

Related Information

- **Recommended HDL Coding Styles documentation** on page 12-1

Multipliers

The Precision Synthesis software detects multipliers in HDL code and maps them directly to device atoms to implement the multiplier in the appropriate type of logic. The Precision Synthesis software also allows you to control the device resources that are used to implement individual multipliers.

Controlling DSP Block Inference for Multipliers

By default, the Precision Synthesis software uses DSP blocks available in Stratix series devices to implement multipliers. The default setting is **AUTO**, which allows the Precision Synthesis software to map to logic look-up tables (LUTs) or DSP blocks, depending on the size of the multiplier. You can use the Precision Synthesis GUI or HDL attributes for direct mapping to only logic elements or to only DSP blocks.

Table 18-3: Options for dedicated_mult Parameter to Control Multiplier Implementation in Precision Synthesis

Value	Description
ON	Use only DSP blocks to implement multipliers, regardless of the size of the multiplier.
OFF	Use only logic (LUTs) to implement multipliers, regardless of the size of the multiplier.
AUTO	Use logic (LUTs) or DSP blocks to implement multipliers, depending on the size of the multipliers.

Setting the Use Dedicated Multiplier Option

To set the `Use Dedicated Multiplier` option in the Precision Synthesis GUI, compile the design, and then in the Design Hierarchy browser, right-click the operator for the desired multiplier and click **Use Dedicated Multiplier**.

Setting the dedicated_mult Attribute

To control the implementation of a multiplier in your HDL code, use the `dedicated_mult` attribute with the appropriate value as shown in the examples below.

Example 18-12: Setting the dedicated_mult Attribute in Verilog HDL

```
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 18-13: Setting the dedicated_mult Attribute in VHDL

```
ATTRIBUTE dedicated_mult: STRING;
ATTRIBUTE dedicated_mult OF <signal name>: SIGNAL IS <value>;
```

The `dedicated_mult` attribute can be applied to signals and wires; it does not work when applied to a register. This attribute can be applied only to simple multiplier code, such as `a = b * c`.

Some signals for which the `dedicated_mult` attribute is set can be removed during synthesis by the Precision Synthesis software for design optimization. In such cases, if you want to force the implementation, you should preserve the signal by setting the `preserve_signal` attribute to `TRUE`.

Example 18-14: Setting the preserve_signal Attribute in Verilog HDL

```
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 18-15: Setting the preserve_signal Attribute in VHDL

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 18-16: Verilog HDL Multiplier Implemented in Logic

```
module unsigned_mult (result, a, b);
   output [15:0] result;
   input [7:0] a;
   input [7:0} b;
   assign result = a * b;
   //synthesis attribute result dedicated_mult OFF
endmodule
```

Example 18-17: VHDL Multiplier Implemented in Logic

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsigned_mult IS
   PORT(
      a: IN std_logic_vector (7 DOWNTO 0);
      b: IN std_logic_vector (7 DOWNTO 0);
      result: OUT std_logic_vector (15 DOWNTO 0));
ATTRIBUTE dedicated_mult: STRING;
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_int, b_int: UNSIGNED (7 downto 0);
   SIGNAL pdt_int: UNSIGNED (15 downto 0);
ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF;
BEGIN
   a_int <= UNSIGNED (a);
   b_int <= UNSIGNED (b);
   pdt_int <= a_int * b_int;
   result <= std_logic_vector(pdt_int);
END rtl;
```

Multiplier-Accumulators and Multiplier-Adders

The Precision Synthesis software also allows you to control the device resources used to implement multiply-accumulators or multiply-adders in your project or in a particular module.

The Precision Synthesis software detects multiply-accumulators or multiply-adders in HDL code and infers an ALTMULT_ACCUM or ALTMULT_ADD IP cores so that the logic can be placed in DSP blocks, or the software maps these functions directly to device atoms to implement the multiplier in the appropriate type of logic.

Note: The Precision Synthesis software supports inference for these functions only if the target device family has dedicated DSP blocks.

For more information about DSP blocks in Altera devices, refer to the appropriate Altera device family handbook and device-specific documentation. For details about which functions a given DSP block can implement, refer to the DSP Solutions Center on the Altera website.

For more information about inferring multiply-accumulator and multiply-adder IP cores in HDL code, refer to the Altera *Recommended HDL Coding Styles* and the Mentor Graphics*Precision Synthesis Style Guide*.

Related Information

Altera DSP Solutions website

Recommended HDL Coding Styles documentation on page 12-1

Controlling DSP Block Inference

By default, the Precision Synthesis software infers the ALTMULT_ADD or ALTMULT_ACCUM IP cores appropriately in your design. These IP cores allow the Quartus II software to select either logic or DSP blocks, depending on the device utilization and the size of the function.

You can use the `extract_mac` attribute to prevent inference of an ALTMULT_ADD or ALTMULT_ACCUM IP cores in a certain module or entity.

Table 18-4: Options for extract_mac Attribute Controlling DSP Implementation

Value	Description
TRUE	The ALTMULT_ADD or ALTMULT_ACCUM IP core is inferred.
FALSE	The ALTMULT_ADD or ALTMULT_ACCUM IP core is not inferred.

To control inference, use the `extract_mac` attribute with the appropriate value from the examples below in your HDL code.

Example 18-18: Setting the extract_mac Attribute in Verilog HDL

```
//synthesis attribute <module name> extract_mac <value>
```

Example 18-19: Setting the extract_mac Attribute in VHDL

```
ATTRIBUTE extract_mac: BOOLEAN;
ATTRIBUTE extract_mac OF <entity name>: ENTITY IS <value>;
```

To control the implementation of the multiplier portion of a multiply-accumulator or multiply-adder, you must use the `dedicated_mult` attribute.

You can use the `extract_mac`, `dedicated_mult`, and `preserve_signal` attributes (in Verilog HDL and VHDL) to implement the given DSP function in logic in the Quartus II software.

Example 18-20: Using extract_mac, dedicated_mult, and preserve_signal in Verilog HDL

```
module unsig_altmult_accuml (dataout, dataa, datab, clk, aclr, clken);
   input [7:0} dataa, datab;
   input clk, aclr, clken;
   output [31:0] dataout;

   reg    [31:0] dataout;
   wire   [15:0] multa;
   wire   [31:0] adder_out;

   assign multa = dataa * datab;

   //synthesis attribute multa preserve_signal TRUE
   //synthesis attribute multa dedicated_mult OFF
   assign adder_out = multa + dataout;

   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      dataout <= 0;
      else if (clken)
      dataout <= adder_out;
   end

   //synthesis attribute unsig_altmult_accuml extract_mac FALSE
endmodule
```

Example 18-21: Using extract_mac, dedicated_mult, and preserve_signal in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
ENTITY signedmult_add IS
   PORT(
      a, b, c, d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   ATTRIBUTE preserve_signal: BOOLEANS;
   ATTRIBUTE dedicated_mult: STRING;
   ATTRIBUTE extract_mac: BOOLEAN;
   ATTRIBUTE extract_mac OF signedmult_add: ENTITY IS FALSE;
END signedmult_add;
ARCHITECTURE rtl OF signedmult_add IS
   SIGNAL a_int, b_int, c_int, d_int : signed (7 DOWNTO 0);
   SIGNAL pdt_int, pdt2_int : signed (15 DOWNTO 0);
   SIGNAL result_int: signed (15 DOWNTO 0);
   ATTRIBUTE preserve_signal OF pdt_int: SIGNAL IS TRUE;
   ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
   ATTRIBUTE preserve_signal OF pdt2_int: SIGNAL IS TRUE;
   ATTRIBUTE dedicated_mult OF pdt2_int: SIGNAL IS "OFF";
```

```
BEGIN
   a_int <= signed (a);
   b_int <= signed (b);
   c_int <= signed (c);
   d_int <= signed (d);
   pdt_int <= a_int * b_int;
   pdt2_int <= c_int * d_int;
   result_int <= pdt_int + pdt2_int;
   result <= STD_LOGIC_VECTOR(result_int);
END rtl;
```

RAM and ROM

The Precision Synthesis software detects memory structures in HDL code and converts them to an operator that infers an ALTSYNCRAM or LPM_RAM_DP IP cores, depending on the device family. The software then places these functions in memory blocks.

The software supports inference for these functions only if the target device family has dedicated memory blocks.

For more information about inferring RAM and ROM IP cores in HDL code, refer to the *Precision Synthesis Style Guide*.

Related Information

- **Recommended HDL Coding Styles documentation** on page 12-1

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to one part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations can be made dramatically faster by focusing new compilations on particular design partitions and merging results with the results of previous compilations of other partitions. You can perform optimization on individual blocks and then integrate them into a final design and optimize the design at the top-level.

The first step in an incremental design flow is to make sure that different parts of your design do not affect each other. You must ensure that you have separate netlists for each partition in your design. If the whole design is in one netlist file, changes in one partition affect other partitions because of possible node name changes when you resynthesize the design.

You can create different implementations for each partition in your Precision Synthesis project, which allows you to switch between partitions without leaving the current project file. You can also create a separate project for each partition if you require separate projects for a team-based design flow. Alternatively, you can use the incremental synthesis capability in the Precision RTL Plus software.

Related Information

- **Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation** on page 3-1

Creating a Design with Precision RTL Plus Incremental Synthesis

The Precision RTL Plus incremental synthesis flow for Quartus II incremental compilation uses a partition-based approach to achieve faster design cycle time.

Using the incremental synthesis feature, you can create different netlist files for different partitions of a design hierarchy within one partition implementation, which makes each partition independent of the others in an incremental compilation flow. Only the portions of a design that have been updated must be recompiled during design iterations. You can make changes and resynthesize one partition in a design to create a new netlist without affecting the synthesis results or fitting of other partitions.

The following steps show a general flow for partition-based incremental synthesis with Quartus II incremental compilation:

1. Create Verilog HDL or VHDL design files.
2. Determine which hierarchical blocks you want to treat as separate partitions in your design, and designate the partitions with the `incr_partition` attribute.
3. Create a project in the Precision RTL Plus Synthesis software and add the HDL design files to the project.
4. Enable incremental synthesis in the Precision RTL Plus Synthesis software using one of these methods:
    - Use the Precision RTL Plus Synthesis GUI to turn on **Enable Incremental Synthesis**.
    - Run the following command in the Transcript Window:

    ```
    setup_design -enable_incr_synth
    ```

5. Run the basic Precision Synthesis flow of compilation, synthesis, and place-and-route on your design. In subsequent runs, the Precision RTL Plus Synthesis software processes only the parts of the design that have changed, resulting in a shorter iteration than the initial run. The performance of the unchanged partitions is preserved.

    The Precision RTL Plus Synthesis software sets the netlist types of the unchanged partitions to **Post Fit** and the changed partitions to **Post Synthesis**. You can change the netlist type during timing closure in the Quartus II software to obtain the best QoR.
6. Import the EDIF or VQM netlist for each partition and the top-level **.tcl** file into the Quartus II software, and set up the Quartus II project to use incremental compilation.
7. Compile your Quartus II project.
8. If you want, you can change the Quartus II incremental compilation netlist type for a partition with the **Design Partitions Window**. You can change the **Netlist Type** to one of the following options:
    - To preserve the previous post-fit placement results, change the **Netlist Type** of the partition to **Post-Fit**.
    - To preserve the previous routing results, set the **Fitter Preservation Level** of the partition to **Placement and Routing**.

Creating Partitions with the incr_partition Attribute

Partitions are set using the HDL `incr_partition` attribute. The Precision Synthesis software creates or deletes partitions by reading this attribute during compilation iterations. The attribute can be attached to either the design unit definition or an instance.

To delete partitions, you can remove the attribute or set the attribute value to false.

Note: The Precision Synthesis software ignores partitions set in a black box.

Example 18-22: Using incr_partition Attribute to Create a Partition in Verilog HDL

```
Design unit partition:

module my_block(
   input clk;
   output reg [31:0] data_out) /* synthesis incr_partition */ ;

Instance partition:

my_block my_block_inst(.clk(clk), .data_out(data_out));
// synthesis attribute my_block_inst incr_partition true
```

Example 18-23: Using incr_partition Attribute to a Create Partition in VHDL

```
Design unit partition:

entity my_block is
   port(
      clk : in std_logic;
      data_out : out std_logic_vector(31 downto 0)
   );
   attribute incr_partition : boolean;
   attribute incr_partition of my_block : entity is true;
end entity my_block;

Instance partition:

component my_block is
   port(
      clk : in std_logic;
      data_out : out std_logic_vector(31 downto 0)
   );
end component;

attribute incr_partition : boolean;
attribute incr_partition of my_block_inst : label is true;

my_block_inst my_block
   port map(clk, data_out);
```

Creating Multiple Mapped Netlist Files With Separate Precision Projects or Implementations

You can manually generate multiple netlist files, which can be VQM or EDIF files, for incremental compilation using black boxes and separate Precision projects or implementations for each design partition. This manual flow is supported in versions of the Precision software that do not include the incremental synthesis feature. You might also use this feature if you perform synthesis in a team-based environment without a top-level synthesis project that includes all of the lower-level design blocks.

In the Precision Synthesis software, create a separate implementation, or a separate project, for each lower-level module and for the top-level design that you want to maintain as a separate netlist file. Implement black box instantiations of lower-level modules in your top-level implementation or project.

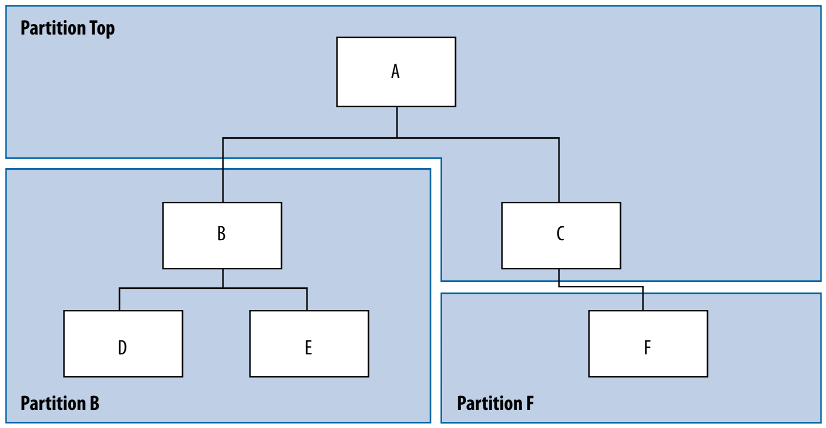
For more information about managing implementations and projects, refer to the *Precision RTL Synthesis User's Manual*.

Note: In a standard Quartus II incremental compilation flow, Precision Synthesis software constraints made on lower-level modules are not passed to the Quartus II software. Ensure that appropriate constraints are made in the top-level Precision Synthesis project, or in the Quartus II project.

Creating Black Boxes to Create Netlists

In the figure below, the top-level partition contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in the sub-block C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers may work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for module B and its submodules D and E, while a third netlist is created for module F.

Figure 18-2: Partitions in a Hierarchical Design



To create multiple EDIF netlist files for this design, follow these steps:

1. Generate a netlist file for module B. Use **B.v**/**.vhd**, **D.v**/**.vhd**, and **E.v**/**.vhd** as the source files.
2. Generate a netlist file for module F. Use **F.v**/**.vhd** as the source file.
3. Generate a top-level netlist file for module A. Use **A.v**/**.vhd** and **C.v**/**.vhd** as the source files. Ensure that you create black boxes for modules B and F, which were optimized separately in the previous steps.

The goal is to individually synthesize and generate a netlist file for each lower-level module and then instantiate these modules as black boxes in the top-level file. You can then synthesize the top-level file to generate the netlist file for the top-level design. Finally, both the lower-level and top-level netlist files are provided to your Quartus II project.

Note: When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate the new netlist file. Do not resynthesize the implementations or projects for the unchanged partitions.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In Verilog HDL, you must provide an empty module declaration for any module that is treated as a black box.

A black box for the top-level file **A.v** is shown in the following example. Provide an empty module declaration for any lower-level files, which also contain a black box for any module beneath the current level of hierarchy.

Example 18-24: Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
   input data_in, clk, e, ld;
   output [15:0] data_out;
   wire [15:0] cnt_out;
   B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
   F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));
   // Any other code in A.v goes here.
endmodule
//Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
boxes.
module B (data_in, clk, ld, data_out);
   input data_in, clk, ld;
   output [15:0] data_out;
endmodule
module F (d, clk, e, q);
   input [15:0] d;
   input clk, e;
   output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In VHDL, you must provide a component declaration for the black box.

A black box for the top-level file **A.vhd** is shown in the example below. Provide a component declaration for any lower-level files that also contain a black box or for any block beneath the current level of hierarchy.

Example 18-25: VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
   PORT ( data_in : IN INTEGER RANGE 0 TO 15;
      clk, e, ld : IN STD_LOGIC;
      data_out : OUT INTEGER RANGE 0 TO 15);
END A;
ARCHITECTURE a_arch OF A IS
COMPONENT B PORT(
   data_in : IN INTEGER RANGE 0 TO 15;
   clk, ld : IN STD_LOGIC;
   d_out : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;
COMPONENT F PORT(
   d : IN INTEGER RANGE 0 TO 15;
   clk, e: IN STD_LOGIC;
   q : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;
```

```
-- Other component declarations in A.vhd go here
signal cnt_out : INTEGER RANGE 0 TO 15;
BEGIN
   U1 : B
   PORT MAP (
      data_in => data_in,
      clk => clk,
      ld => ld,
      d_out => cnt_out);
   U2 : F
   PORT MAP (
      d => cnt_out,
      clk => clk,
      e => e,
      q => data_out);
   -- Any other code in A.vhd goes here
END a_arch;
```

After you complete the steps outlined above, you have different netlist files for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

Creating Quartus II Projects for Multiple Netlist Files

The Precision Synthesis software creates a **.tcl** file for each implementation, and provides the Quartus II software with the appropriate constraints and information to set up a project. When using incremental synthesis, the Precision RTL Plus Synthesis software creates only a single **.tcl** file, <*project name*>**_incr_partitions.tcl**, to pass the partition information to the Quartus II software.

Depending on your design methodology, you can create one Quartus II project for all netlists, or a separate Quartus II project for each netlist. In the standard incremental compilation design flow, you create design partition assignments for each partition in the design within a single Quartus II project. This methodology provides the best QoR and performance preservation during incremental changes to your design. You might require a bottom-up design flow if each partition must be optimized separately, such as for third-party IP delivery.

To follow this design flow in the Quartus II software, create separate Quartus II projects and export each design partition and incorporate it into a top-level design using the incremental compilation features to maintain placement results.

Related Information

Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script on page 18-10

Creating a Single Quartus II Project for a Standard Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file generated for the top-level partition to create your Quartus II project and import all the netlists into this one Quartus II project for an incremental compilation flow. You can optimize all partitions within the single Quartus II project and take advantage of the performance preservation and compilation time reduction that incremental compilation provides.

All the constraints from the top-level implementation are passed to the Quartus II software in the top-level **.tcl** file, but any constraints made only in the lower-level implementations within the Precision Synthesis software are not forward-annotated. Enter these constraints manually in your Quartus II project.

Creating Multiple Quartus II Projects for a Bottom-Up Flow

Use the **.tcl** files generated by the Precision Synthesis software for each Precision Synthesis software implementation or project to generate multiple Quartus II projects, one for each partition in the design. Each designer in the project can optimize their block separately in the Quartus II software and export the placement of their blocks using incremental compilation. Designers should create a LogicLock region to provide a floorplan location assignment for each block; the top-level designer should then import all the blocks and assignments into the top-level project.

Hierarchy and Design Considerations

To ensure the proper functioning of the synthesis flow, you can create separate partitions only for modules, entities, or existing netlist files. In addition, each module or entity must have its own design file. If two different modules are in the same design file, but are defined as being part of different partitions, incremental synthesis cannot be maintained because both regions must be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Precision Synthesis software pushes the tri-states through the hierarchy to the top-level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

Related Information

- **Best Practices for Incremental Compilation Partitions and Floorplan Assignments documentation** on page 14-1

Document Revision History

Table 18-5: Document Revision History

Date	Version	Changes
June 2014	14.0.0	• Dita conversion. • Removed obsolete devices. • Replaced Megafunction, MegaWizard, and IP Toolbench content with IP Catalog and Parameter Editor content.
June 2012	12.0.0	• Removed survey link.
November 2011	10.1.1	• Template update. • Minor editorial changes.

Date	Version	Changes
December 2010	10.1.0	• Changed to new document template. • Removed Classic Timing Analyzer support. • Added support for **. vqm** netlist files. • Edited the "Creating Quartus II Projects for Multiple EDIF Files" on page 15–30 section for changes with the incremental compilation flow. • Editorial changes.
July 2010	10.0.0	• Minor updates for the Quartus II software version 10.0 release
November 2009	9.1.0	• Minor updates for the Quartus II software version 9.1 release
March 2009	9.0.0	• Updated list of supported devices for the Quartus II software version 9.0 release • Chapter 11 was previously Chapter 10 in software version 8.1

Date	Version	Changes
November 2008	8.1.0	• Changed to 8-1/2 x 11 page size • Title changed to *Mentor Graphics Precision Synthesis Support* • Updated list of supported devices • Added information about the Precision RTL Plus incremental synthesis flow • Updated Figure 10-1 to include SystemVerilog • Updated "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 10–19 • Updated "Incremental Compilation and Block-Based Design" on page 10–28 • Added section "Creating Partitions with the incr_partition Attribute" on page 10–29
May 2008	8.0.0	• Removed Mercury from the list of supported devices • Changed Precision version to 2007a update 3 • Added note for Stratix IV support • Renamed "Creating a Project and Compiling the Design" section to "Creating and Compiling a Project in the Precision RTL Synthesis Software" • Added information about constraints in the Tcl file • Updated document based on the Quartus II software version 8.0

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

Related Information

Quartus II Handbook Archive

Analyzing Designs with Quartus II Netlist Viewers 19

2014.06.30

QII5V1 Subscribe Send Feedback

This chapter describes how you can use the Quartus® II Netlist Viewers to analyze and debug your designs.

As FPGA designs grow in size and complexity, the ability to analyze, debug, optimize, and constrain your design is critical. With today's advanced designs, several design engineers are involved in coding and synthesizing different design blocks, making it difficult to analyze and debug the design. The Quartus II RTL Viewer, State Machine Viewer, and Technology Map Viewer provide powerful ways to view your initial and fully mapped synthesis results during the debugging, optimization, and constraint entry processes.

This chapter contains the following sections:

Related Information

When to Use the Netlist Viewers: Analyzing Design Problems

You can use the Netlist Viewers to analyze and debug your design. This section provides simple examples of how to use the RTL Viewer, State Machine Viewer, and Technology Map Viewer to analyze problems encountered in the design process.

Using the RTL Viewer is a good way to view your initial synthesis results to determine whether you have created the necessary logic, and that the logic and connections have been interpreted correctly by the software. You can use the RTL Viewer and State Machine Viewer to check your design visually before simulation or other verification processes. Catching design errors at this early stage of the design process can save you valuable time.

If you see unexpected behavior during verification, use the RTL Viewer to trace through the netlist and ensure that the connections and logic in your design are as expected. You can also view state machine transitions and transition equations with the State Machine Viewer. Viewing your design helps you find and analyze the source of design problems. If your design looks correct in the RTL Viewer, you know to focus your analysis on later stages of the design process and investigate potential timing violations or issues in the verification flow itself.

You can use the Technology Map Viewer to look at the results at the end of Analysis and Synthesis. If you have compiled your design through the Fitter stage, you can view your post-mapping netlist in the Technology Map Viewer (Post-Mapping) and your post-fitting netlist in the Technology Map Viewer. If you perform only Analysis and Synthesis, both the Netlist Viewers display the same post-mapping netlist.

In addition, you can use the RTL Viewer or Technology Map Viewer to locate the source of a particular signal, which can help you debug your design. Use the navigation techniques described in this chapter to search easily through your design. You can trace back from a point of interest to find the source of the signal and ensure the connections are as expected.

The Technology Map Viewer can help you locate post-synthesis nodes in your netlist and make assignments when optimizing your design. This functionality is useful when making a multicycle clock timing assignment between two registers in your design. Start at an I/O port and trace forward or backward through the design and through levels of hierarchy to find nodes of interest, or locate a specific register by visually inspecting the schematic.

You can use the RTL Viewer, State Machine Viewer, and Technology Map Viewer in many other ways throughout the design, debug, and optimization stages. This chapter shows you how to use the various features of the Netlist Viewers to increase your productivity when analyzing a design.

Related Information

- **Quartus II Design Flow with the Netlist Viewers** on page 19-2
- **State Machine Viewer Overview** on page 19-4
- **RTL Viewer Overview** on page 19-3
- **Technology Map Viewer Overview** on page 19-5

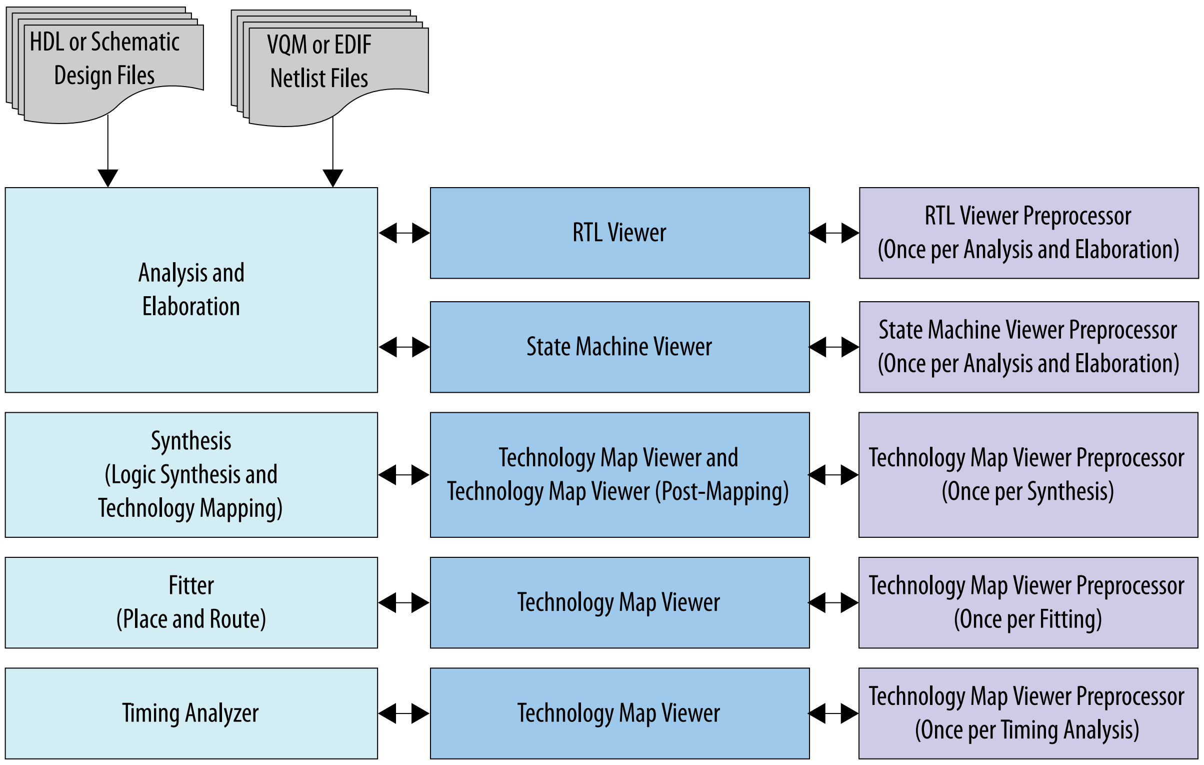
Quartus II Design Flow with the Netlist Viewers

When you first open one of the Netlist Viewers after compiling the design, a preprocessor stage runs automatically before the Netlist Viewer opens.

The preprocessor process box contains a link to the **Settings** > **Compilation Process Settings** page where you can turn on the **Run Netlist Viewers preprocessing during compilation** option. When this option is turned on, the preprocessing becomes part of the full project compilation flow and the Netlist Viewer opens immediately without displaying the preprocessing dialog.

Figure 19-1: Quartus II Design Flow Including the RTL Viewer and Technology Map Viewer

This figure shows how Netlist Viewers fit into the basic Quartus II design flow.


HDL or Schematic Design Files
VQM or EDIF Netlist Files
Analysis and Elaboration
RTL Viewer
RTL Viewer Preprocessor (Once per Analysis and Elaboration)
State Machine Viewer
State Machine Viewer Preprocessor (Once per Analysis and Elaboration)
Synthesis (Logic Synthesis and Technology Mapping)
Technology Map Viewer and Technology Map Viewer (Post-Mapping)
Technology Map Viewer Preprocessor (Once per Synthesis)
Fitter (Place and Route)
Technology Map Viewer
Technology Map Viewer Preprocessor (Once per Fitting)
Timing Analyzer
Technology Map Viewer
Technology Map Viewer Preprocessor (Once per Timing Analysis)

Before the Netlist Viewer can run the preprocessor stage, you must compile your design:

- To open the RTL Viewer or State Machine Viewer, first perform Analysis and Elaboration.
- To open the Technology Map Viewer (Post-Fitting) or the Technology Map Viewer (Post-Mapping), first perform Analysis and Synthesis.

The Netlist Viewers display the results of the last successful compilation. Therefore, if you make a design change that causes an error during Analysis and Elaboration, you cannot view the netlist for the new design files, but you can still see the results from the last successfully compiled version of the design files. If you receive an error during compilation and you have not yet successfully run the appropriate compilation stage for your project, the Netlist Viewer cannot be displayed; in this case, the Quartus II software issues an error message when you try to open the Netlist Viewer.

Note: If the Netlist Viewer is open when you start a new compilation, the Netlist Viewer closes automatically. You must open the Netlist Viewer again to view the new design netlist after compilation completes successfully.

RTL Viewer Overview

The Quartus II RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your Quartus II integrated synthesis results or your third-party netlist file in the Quartus II software.

You can view results after Analysis and Elaboration when your design uses any supported Quartus II design entry method, including Verilog HDL Design Files (**.v**), SystemVerilog Design Files (**. sv**), VHDL Design Files (**. vhd**), AHDL Text Design Files (**.tdf**), or schematic Block Design Files (**.bdf**). You can also view the hierarchy of atom primitives (such as device logic cells and I/O ports) when your design uses a synthesis tool to generate a Verilog Quartus Mapping File (**.vqm**) or Electronic Design Interchange Format (**.edf**) file.

The Quartus II RTL Viewer displays a schematic view of the design netlist after Analysis and Elaboration or netlist extraction is performed by the Quartus II software, but before technology mapping and any synthesis or fitter optimizations. This view is not the final design structure because optimizations have not yet occurred. This view most closely represents your original source design. If you synthesized your design with the Quartus II integrated synthesis, this view shows how the Quartus II software interpreted your design files. If you use a third-party synthesis tool, this view shows the netlist written by your synthesis tool.

When displaying your design, the RTL Viewer optimizes the netlist to maximize readability in the following ways:

- Logic with no fan-out (its outputs are unconnected) and logic with no fan-in (its inputs are unconnected) are removed from the display.
- Default connections such as V_{CC} and GND are not shown.
- Pins, nets, wires, module ports, and certain logic are grouped into buses where appropriate.
- Constant bus connections are grouped.
- Values are displayed in hexadecimal format.
- NOT gates are converted to bubble inversion symbols in the schematic.
- Chains of equivalent combinational gates are merged into a single gate. For example, a 2-input AND gate feeding a 2-input AND gate is converted to a single 3-input AND gate.
- State machine logic is converted into a state diagram, state transition table, and state encoding table, which are displayed in the State Machine Viewer.

To run the RTL Viewer for a Quartus II project, first analyze the design to generate an RTL netlist. To analyze the design and generate an RTL netlist, on the Processing menu, point to **Start** and click **Start Analysis & Elaboration**. You can also perform a full compilation on any process that includes the initial Analysis and Elaboration stage of the Quartus II compilation flow.

To run the RTL Viewer, on the Tools menu, point to **Netlist Viewers** and click **RTL Viewer**.

State Machine Viewer Overview

The State Machine Viewer presents a high-level view of finite state machines in your design. The State Machine Viewer provides a graphical representation of the states and their related transitions, as well as a state transition table that displays the condition equation for each of the state transitions, and encoding information for each state.

To run the State Machine Viewer, on the Tools menu, point to **Netlist Viewers** and click **State Machine Viewer**. To open the State Machine Viewer for a particular state machine, double-click the state machine instance in the RTL Viewer.

Related Information

State Machine Viewer on page 19-20

Technology Map Viewer Overview

The Quartus II Technology Map Viewer provides a technology-specific, graphical representation of your design after Analysis and Synthesis or after the Fitter has mapped your design into the target device.

The Technology Map Viewer shows the hierarchy of atom primitives (such as device logic cells and I/O ports) in your design. For supported families, you can also view internal registers and look-up tables (LUTs) inside logic cells (LCELLs) and registers in I/O atom primitives.

Where possible, the port names of each hierarchy are maintained throughout synthesis; however, port names might change or be removed from the design. For example, if a port is unconnected or driven by GND or V_{CC}, it is removed during synthesis. When a port name changes, the port is assigned a related user logic name in the design or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, on the Processing menu, point to **Start** and click **Start Analysis & Synthesis** to synthesize and map the design to the target technology. At this stage, the Technology Map Viewer shows the same post-mapping netlist as the Technology Map Viewer (Post-Mapping). You can also perform a full compilation, or any process that includes the synthesis stage in the compilation flow.

If you have completed the Fitter stage, the Technology Map Viewer shows the changes made to your netlist by the Fitter, such as physical synthesis optimizations, while the Technology Map Viewer (Post-Mapping) shows the post-mapping netlist. If you have completed the Timing Analysis stage, you can locate timing paths from the Timing Analyzer report in the Technology Map Viewer.

To open the Technology Map Viewer, on the Tools menu, point to **Netlist Viewers** and click **Technology Map Viewer (Post-Fitting)** or **Technology Map Viewer (Post Mapping)**.

Related Information

- **View Contents of Nodes in the Schematic View** on page 19-16
- **Viewing a Timing Path** on page 19-23

Introduction to the User Interface

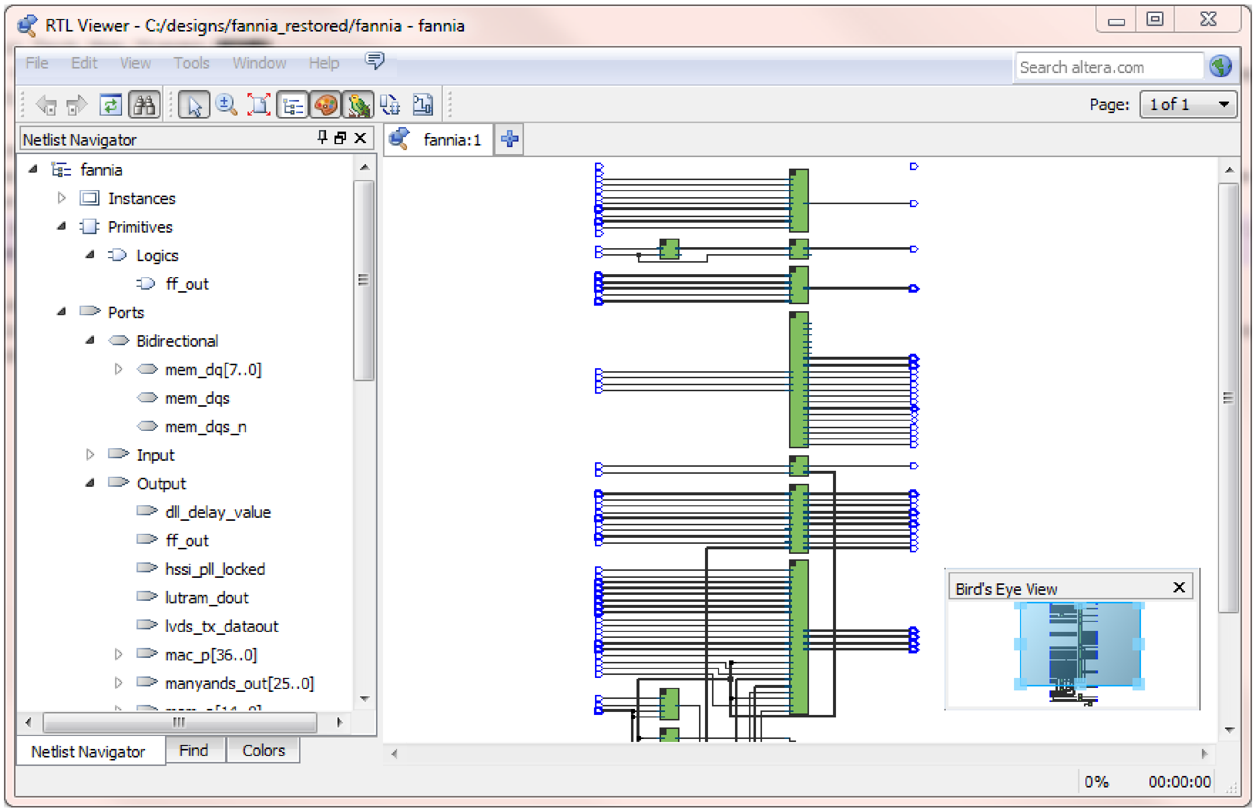
The Netlist Viewer is a graphical user-interface for viewing and manipulating nodes and nets in the netlist.

The RTL Viewer and Technology Map Viewer each consist of these main parts:

- The **Netlist Navigator** pane—displays a representation of the project hierarchy.
- The **Find** pane—allows you to find and locate specific design elements in the schematic view.
- The **Properties** pane displays the properties of the selected block when you select **Properties** from the shortcut menu.
- The schematic view—displays a graphical representation of the internal structure of your design.

Figure 19-2: RTL Viewer

This figure shows the schematic view and the **Netlist Navigator** pane of the RTL Viewer.



Netlist Viewers also contain a toolbar that provides tools to use in the schematic view.

- Use the **Back** and **Forward** buttons to switch between schematic views. You can go forward only if you have not made any changes to the view since going back. These commands do not undo an action, such as selecting a node. The Netlist Viewer caches up to ten actions including filtering, hierarchy navigation, netlist navigation, and zooming.
- The **Refresh** button restores the schematic view and optimizes the layout. **Refresh** does not reload the database if you change your design and recompile.
- The **Find** button opens and closes the **Find** pane.
- The **Selection tool** and **Zoom tool** buttons toggle between the selection mode and zoom mode.
- The **Fit in Page** button resets the schematic view to encompass the entire design.
- The **Netlist Navigator** button opens or closes the **Netlist Navigator** pane.
- The **Color Settings** button opens the **Colors** pane where you can customize the color scheme used in the Netlist Viewer.

- The **Bird's Eye View** button opens the **Bird's Eye View** window which displays a miniature version of your design and allows you to navigate within the design and adjust the magnification in the schematic view quickly.
- The **Show/Hide Instance Pins** button can toggle the display of instance pins not displayed by functions such as cross-probing between a Netlist Viewer and TimeQuest. You can also use it to hide unconnected instance pins when filtering a node results in large numbers of unconnected or unused pins.
- The **Show Netlist on One Page** button displays the netlist on a single page if the Netlist Veiwer has split the design across several pages. This can make netlist tracing easier.

You can have only one RTL Viewer, one Technology Map Viewer (Post-Fitting), one Technology Map Viewer (Post-Mapping), and one State Machine Viewer window open at the same time, although each window can show multiple pages, each with multiple tabs. For example, you cannot have two RTL Viewer windows open at the same time.

Related Information

- Netlist Navigator Pane on page 19-7
- Netlist Viewers Find Pane on page 19-10
- Properties Pane on page 19-8

Netlist Navigator Pane

The **Netlist Navigator** pane displays the entire netlist in a tree format based on the hierarchical levels of the design. In each level, similar elements are grouped into subcategories.

You can use the **Netlist Navigator** pane to traverse through the design hierarchy to view the logic schematic for each level. You can also select an element in the **Netlist Navigator** to highlight in the schematic view.

Note: Nodes inside atom primitives are not listed in the **Netlist Navigator** pane.

For each module in the design hierarchy, the **Netlist Navigator** pane displays the applicable elements listed in the following table. Click the “+” icon to expand an element.

Table 19-1: Netlist Navigator Pane Elements

Elements	Description
Instances	Modules or instances in the design that can be expanded to lower hierarchy levels.
State Machines	State machine instances in the design that can be viewed in the State Machine Viewer.

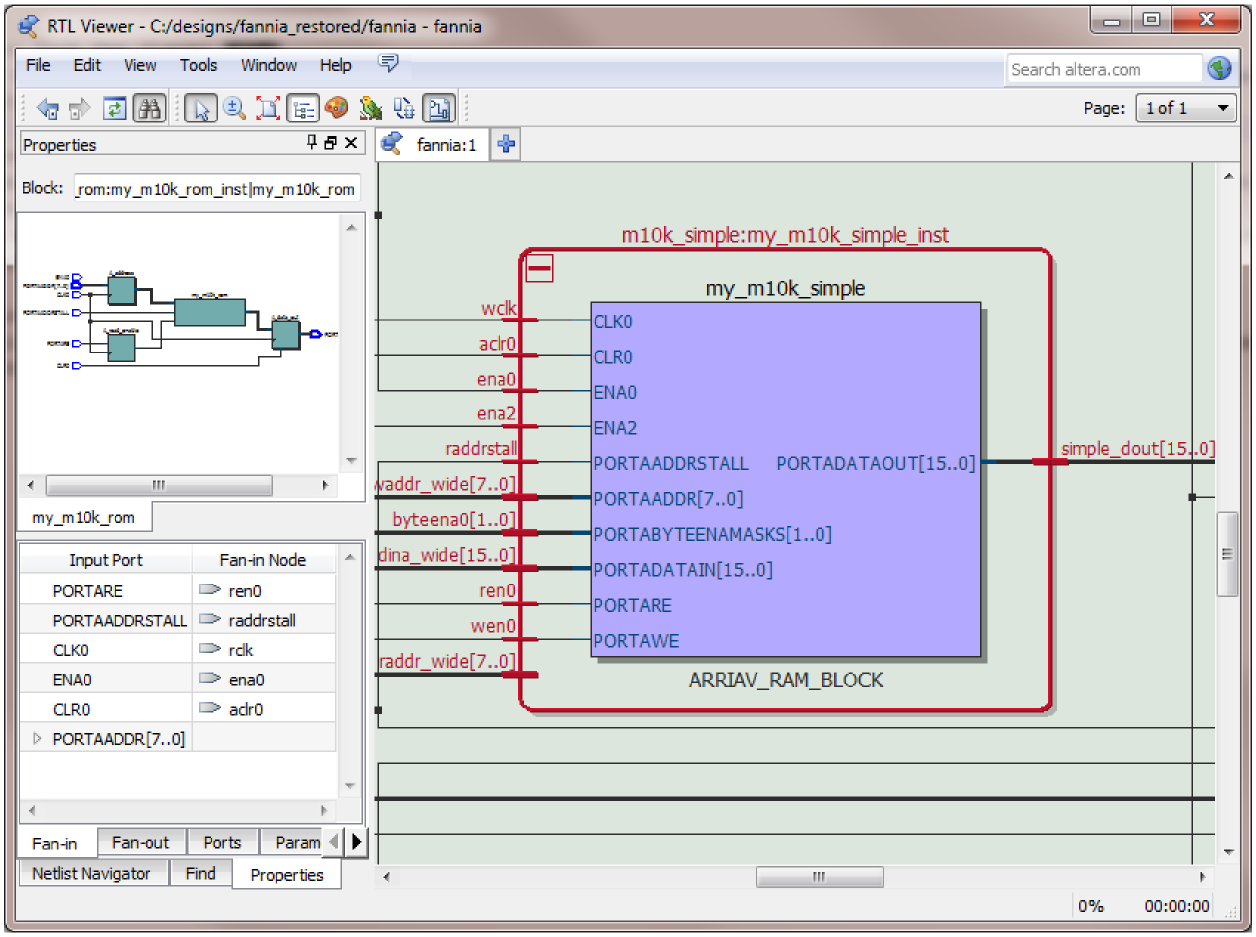
Elements	Description
Primitives	Low-level nodes that cannot be expanded to any lower hierarchy level. These primitives include: • Registers and gates that you can view in the RTL Viewer when using Quartus II integrated synthesis • Logic cell atoms in the Technology Map Viewer or in the RTL Viewer when using a VQM or EDIF from third-party synthesis software In the Technology Map Viewer, you can view the internal implementation of certain atom primitives, but you cannot traverse into a lower-level of hierarchy.
Ports	The I/O ports in the current level of hierarchy. • Pins are device I/O pins when viewing the top hierarchy level and are I/O ports of the design when viewing the lower-levels. • When a pin represents a bus or an array of pins, expand the pin entry in the list view to see individual pin names.

Properties Pane

You can view the properties of an instance or primitive using the **Properties** pane.

Figure 19-3: Properties Pane

To view the properties of an instance or primitive in the RTL Viewer or Technology Map Viewer, right-click the node and click **Properties**.



Input Port	Fan-in Node
PORTARE	ren0
PORTAADDRSTALL	raddrstall
CLK0	rclk
ENA0	ena0
CLR0	aclr0
PORTAADDR[7..0]	

The **Properties** pane contains tabs with the following information about the selected node:

- The **Fan-in** tab displays the **Input port** and **Fan-in Node**.
- The **Fan-out** tab displays the **Output port** and **Fan-out Node**.
- The **Parameters** tab displays the **Parameter Name** and **Values** of an instance.
- The **Ports** tab displays the **Port Name** and **Constant** value (for example, V_{CC} or GND). The possible value of a port are listed below.

Table 19-2: Possible Port Values

Value	Description
V_{CC}	The port is not connected and has V_{CC} value (tied to V_{CC})
GND	The port is not connected and has GND value (tied to GND)

Value	Description
--	The port is connected and has value (other than V_{CC} or GND)
Unconnected	The port is not connected and has no value (hanging)

If the selected node is an atom primitive, the **Properties** pane displays a schematic of the internal logic.

Netlist Viewers Find Pane

You can narrow the range of the search process by setting the following options in the **Find** pane:

- Click **Browse** in the **Find** pane to specify the hierarchy level of the search. In the **Select Hierarchy Level** dialog box, select the particular instance you want to search.
- Turn on the **Include subentities** option to include child hierarchies of the parent instance during the search.
- Click **Options** to open the **Find Options** dialog box. Turn on **Instances**, **Nodes**, **Ports**, or any combination of the three to further refine the parameters of the search.

When you click the **List** button, a progress bar appears below the **Find** box.

All results that match the criteria you set are listed in a table. When you double-click an item in the table, the related node is highlighted in red in the schematic view.

Schematic View

The schematic view is shown on the right side of the RTL Viewer and Technology Map Viewer. The schematic view contains a schematic representing the design logic in the netlist. This view is the main screen for viewing your gate-level netlist in the RTL Viewer and your technology-mapped netlist in the Technology Map Viewer.

The RTL Viewer and Technology Map Viewer attempt to display schematic in a single page view by default. If the schematic crosses over to several pages, you can highlight a net and use connectors to trace the signal in a single page.

Display Schematics in Multiple Tabbed View

The RTL Viewer and Technology Map Viewer support multiple tabbed views.

With multiple tabbed view, schematics can be displayed in different tabs. Selection is independent between tabbed views, but selection in the tab in focus is synchronous with the Netlist Navigator pane.

To create a new blank tab, click the **New Tab** button at the end of the tab row . You can now drag a node from the **Netlist Navigator** pane into the schematic view.

You can right-click in a tab to see a shortcut menu where you can:

- Create a blank view with **New Tab**
- Create a **Duplicate Tab** of the tab in focus
- Choose to **Cascade Tabs**
- Choose to**Tile Tabs**
- Choose **Close Tab** to close the tab in focus
- Choose **Close Other Tabs** to close all tabs except the tab in focus

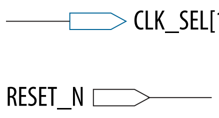
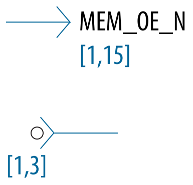
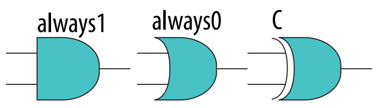
Schematic Symbols

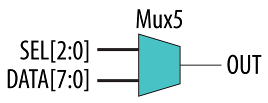
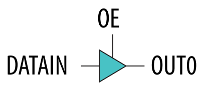
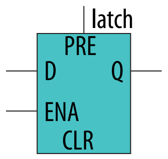
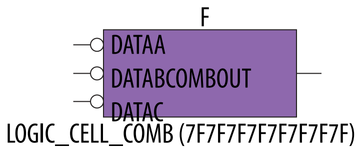
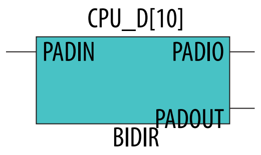
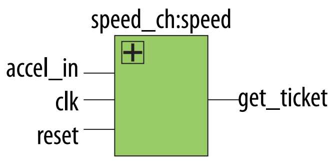
The symbols for nodes in the schematic represent elements of your design netlist. These elements include input and output ports, registers, logic gates, Altera® primitives, high-level operators, and hierarchical instances

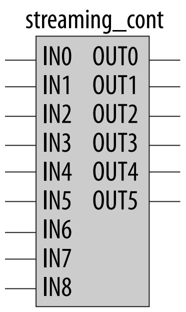
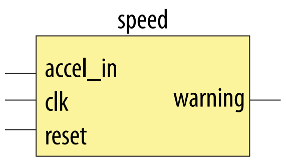
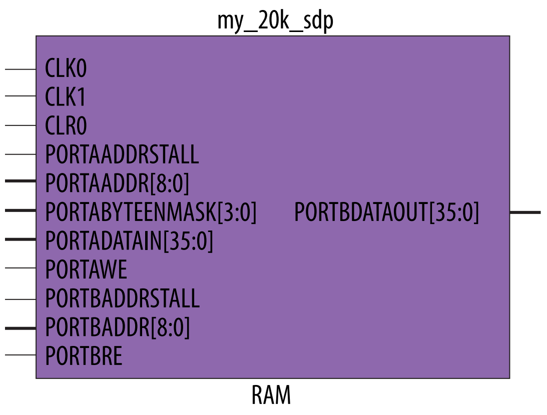
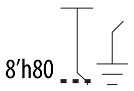
The following table lists and describes the primitives and basic symbols that you can display in the schematic view of the RTL Viewer and Technology Map Viewer.

Note: The logic gates and operator primitives appear only in the RTL Viewer. Logic in the Technology Map Viewer is represented by atom primitives, such as registers and LCELLs.

Table 19-3: Symbols in the Schematic View

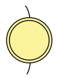
Symbol	Description
I/O Ports 	An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can also represent a bus. Only one wire is shown connected to the bidirectional symbol, representing the input and output paths. Input symbols appear on the left-most side of the schematic. Output and bidirectional symbols appear on the right-most side of the schematic.
I/O Connectors 	An input or output connector, representing a net that comes from another page of the same hierarchy. To go to the page that contains the source or the destination, double-click on the connector to jump to the appropriate page.
OR, AND, XOR Gates 	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output port indicates the port is inverted.

Symbol	Description
MULTIPLEXER 	A multiplexer primitive with a selector port that selects between port `0` and port `1`. A multiplexer with more than two inputs is displayed as an operator.
BUFFER 	A buffer primitive. The figure shows the tri-state buffer, with an inverted output enable port. Other buffers without an enable port include LCELL, SOFT, CARRY, and GLOBAL. The NOT gate and EXP expander buffers use this symbol without an enable port and with an inverted output port.
LATCH 	A latch/DFF (data flipflop) primitive. A DFF has the same ports as a latch and a clock trigger. The other flipflop primitives are similar: • DFFEA (data flipflop with enable and asynchronous load) primitive with additional `ALOAD` asynchronous load and `ADATA` data signals • DFFEAS (data flipflop with enable and synchronous and asynchronous load), which has `ASDATA` as the secondary data port
Atom Primitive  F DATAA DATABCOMBOUT DATAC LOGIC_CELL_COMB (7F7F7F7F7F7F7F7F)	An atom primitive. The symbol displays the atom name, the port names, and the atom type. The blue shading indicates an atom primitive for which you can view the internal details.
Other Primitive 	Any primitive that does not fall into the previous categories. Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive or operator type, and its name.
Instance 	An instance in the design that does not correspond to a primitive or operator (a user-defined hierarchy block). The symbol displays the port name and the instance name.

Symbol	Description
Ecrypted Instance  streaming_cont IN0 IN1 IN2 IN3 IN4 IN5 IN6 IN7 IN8 OUT0 OUT1 OUT2 OUT3 OUT4 OUT5	A user-defined encrypted instance in the design. The symbol displays the instance name. You cannot open the schematic for the lower-level hierarchy, because the source design is encrypted.
State Machine Instance 	A finite state machine instance in the design.
RAM  my_20k_sdp CLK0 CLK1 CLR0 PORTAADDRSTALL PORTAADDR[8:0] PORTABYTEENMASK[3:0] PORTADATAIN[35:0] PORTAWE PORTBADDRSTALL PORTBADDR[8:0] PORTBRE PORTBDATAOUT[35:0] RAM	A synchronous memory instance with registered inputs and optionally registered outputs. The symbol shows the device family and the type of memory block. This figure shows a true dual-port memory block in a Stratix M-RAM block.
Constant 	A constant signal value that is highlighted in gray and displayed in hexadecimal format by default throughout the schematic.

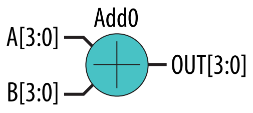
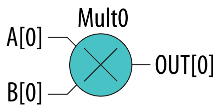
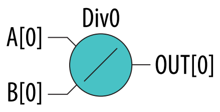
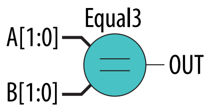
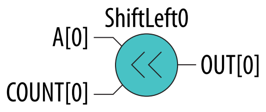
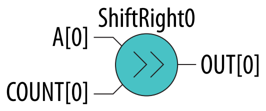
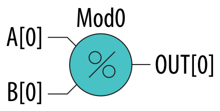
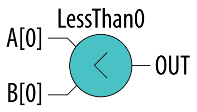
The following table lists and describes the symbol open only in the State Machine Viewer.

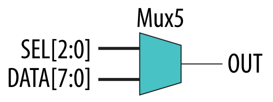
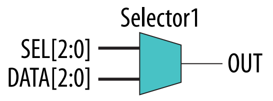
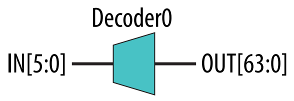
Table 19-4: Symbol Available Only in the State Machine Viewer

Symbol	Description
State Node 	The node representing a state in a finite state machine. State transitions are indicated with arcs between state nodes. The double circle border indicates the state connects to logic outside the state machine, and a single circle border indicates the state node does not feed outside logic.

The following lists and describes the additional higher level operator symbols in the RTL Viewer schematic view.

Table 19-5: Operator Symbols in the RTL Viewer Schematic View

Symbol	Description
	An adder operator: `OUT = A + B`
	A multiplier operator: `OUT = A ¥ B`
	A divider operator: `OUT = A / B`
	Equals
	A left shift operator: `OUT = (A << COUNT)`
	A right shift operator: `OUT = (A >> COUNT)`
	A modulo operator: `OUT = (A%B)`
	A less than comparator: `OUT = (A<:B:A>B)`

Symbol	Description
	A multiplexer: `OUT = DATA [SEL]` The data range size is $2^{\text{sel range size}}$
	A selector: A multiplexer with one-hot select input and more than two input signals
	A binary number decoder: `OUT` = (binary_number (`IN`) == x) for x = 0 to x = $2^{(n+1)}$ - 1

Related Information

- **Partition the Schematic into Pages** on page 19-19
- **Follow Nets Across Schematic Pages** on page 19-19
- **State Machine Viewer** on page 19-20

Select Items in the Schematic View

To select an item in the schematic view, ensure that the **Selection Tool** is enabled in the Netlist Viewer toolbar (this tool is enabled by default). Click an item in the schematic view to highlight it in red.

Select multiple items by pressing the Shift key while selecting with your mouse.

Items selected in the schematic view are automatically selected in the **Netlist Navigator** pane. The folder then expands automatically if it is required to show the selected entry; however, the folder does not collapse automatically when you are not using or you have deselected the entries.

When you select a hierarchy box, node, or port in the schematic view, the item is highlighted in red but none of the connecting nets are highlighted. When you select a net (wire or bus) in the schematic view, all connected nets are highlighted in red.

Once you have selected an item, you can perform different actions on it based on the contents of the shortcut menu which appears when you right-click on your selection.

Related Information

Netlist Navigator Pane on page 19-7

Shortcut Menu Commands in the Schematic View

When you right-click on an instance or primitive selected in the schematic view, the Netlist Viewer displays a shortcut menu.

If the selected item is a node, you see the following options:

- Click **Expand to Upper Hierarchy** to displays the parent hierarchy of the node in focus.
- Click **Copy ToolTip** to copy the selected item name to the clipboard. This command does not work on nets.
- Click **Hide Selection** to remove the selected item from the schematic view. This does not delete the item from the design, merely masks it in the current view.
- Click **Filtering** to display a sub-menu with options for filtering your selection.

When the selected item is a net, the shortcut menu displays the option to **Unbundle Net**. When you unbundle a net, all connected bus pins and ports are ungrouped and displayed. You can use this to trace bundled connections more easily.

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only the logic elements of interest to you.

You can filter your netlist by selecting hierarchy boxes, nodes, ports of a node, or states in a state machine that are part of the path you want to see. The following filter commands are available:

- **Sources**—Displays the sources of the selection.
- **Destinations**—Displays the destinations of the selection.
- **Sources & Destinations**—displays the sources and destinations of the selection.
- **Selected Nodes**—Displays only the selected nodes.
- **Between Selected Nodes**—Displays nodes and connections in the path between the selected nodes .
- **Bus Index**—Displays the sources or destinations for one or more indices of an output or input bus port .
- **Filtering Options**—Displays the **Filtering Options** dialog box:
    - **Stop filtering at register**—Turning this option on directs the Netlist Viewer to filter out to the nearest register boundary.
    - **Filter across hierarchies**—Turning this option on directs the Netlist Viewer to filter across hierarchies.
    - **Maximum number of hierarchy levels**—Sets the maximium number of hierarchy levels displayed in the schematic view.

To filter your netlist, select a hierarchy box, node, port, net, or state node, right-click in the window, point to **Filter** and click the appropriate filter command. The Netlist Viewer generates a new page showing the netlist that remains after filtering.

When filtering in a state diagram in the State Machine Viewer, sources and destinations refer to the previous and next transition states or paths between transition states in the state diagram. The transition table and encoding table also reflect the filtering.

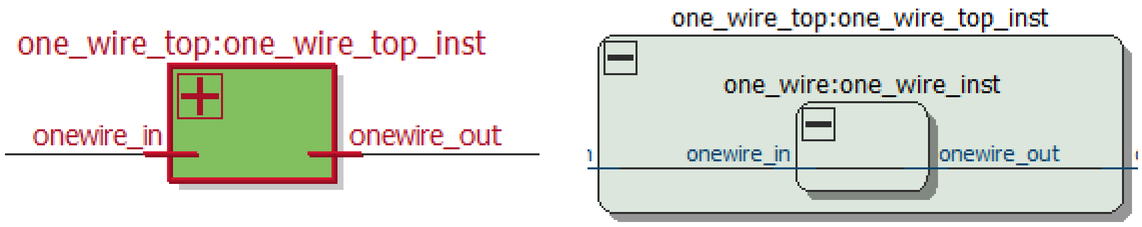
View Contents of Nodes in the Schematic View

In the RTL Viewer and the Technology Map Viewer, you can view the contents of nodes to see their underlying implementation details.

You can view LUTs, registers, and logic gates. You can also view the implementation of RAM and DSP blocks in certain devices in the RTL Viewer or Technology Map Viewer. In the Technology Map Viewer, you can view the contents of primitives to see their underlying implementation details.

Figure 19-4: Wrapping and Unwrapping Objects

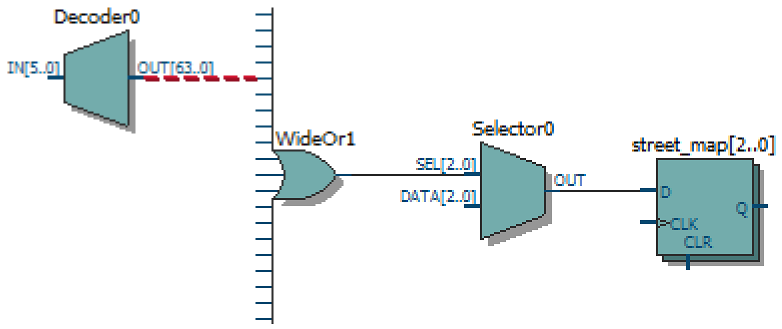
If you can unwrap the contents of an instance, a plus symbol appears in the upper right corner of the object in the schematic view. To wrap the contents (and revert to the compact format), click the minus symbol in the upper right corner of the unwrapped instance.



Note: In the schematic view, the internal details in an atom instance cannot be selected as individual nodes. Any mouse action on any of the internal details is treated as a mouse action on the atom instance.

Figure 19-5: Nodes with Connections Outside the Hierarchy

In some cases, the selected instance connects to something outside the visible level of the hierarchy in the schematic view. In this case, the net appears as a dotted line. Double-click on the dotted line to expand the view to display the destination of the connection .

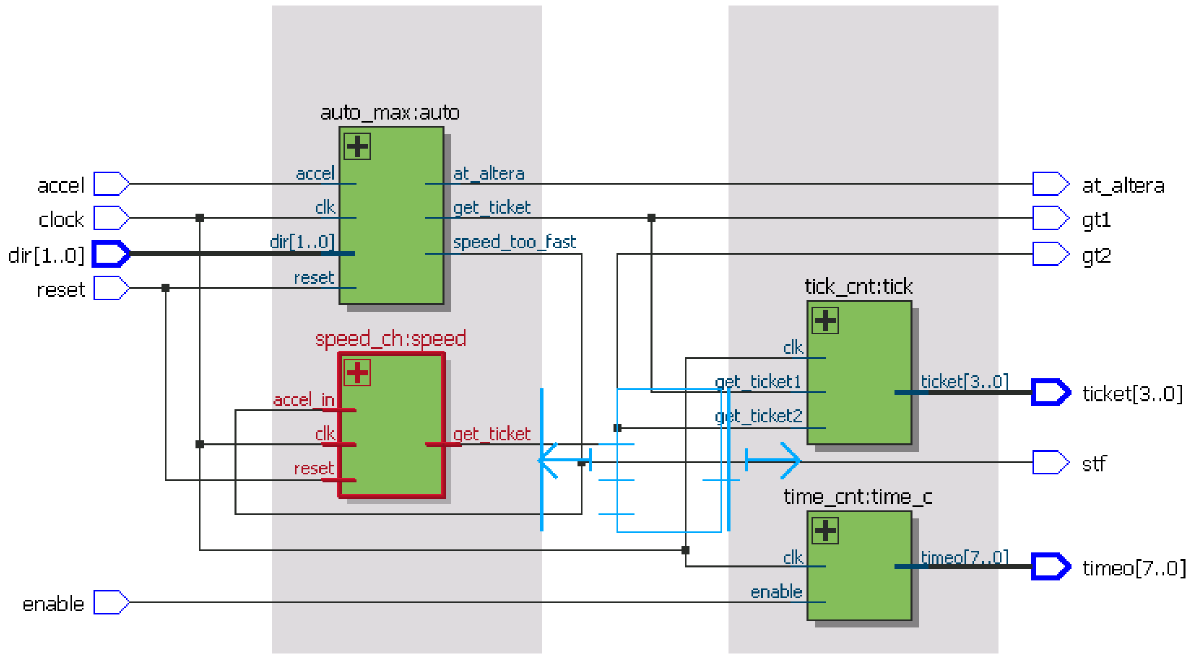


Moving Nodes in the Schematic View

You can drag and drop items in the schematic view to rearrange them.

Figure 19-6: Drag and Drop Movement of Nodes

To move a node from one area of the netlist to another, select the node and hold down the Shift key. Legal placements appear as shaded areas within the hierarchy. Click to drop the selected node.



Right-click and click on **Refresh** to restore the schematic view to its default arrangement.

View LUT Representations in the Technology Map Viewer

You can view different representations of a LUT by right-clicking the selected LUT and clicking **Properties**.

You can view the LUT representations in the following three tabs in the **Properties** dialog box:

- The **Schematic** tab—the equivalent gate representations of the LUT.
- The **Truth Table** tab—the truth table representations.

Related Information

Properties Pane on page 19-8

Zoom Controls

You can control the magnification of your schematic on the View menu, with the Zoom Tool in the toolbar, or with mouse gestures.

By default, the Netlist Viewer displays most pages sized to fit in the window. If the schematic page is very large, the schematic is displayed at the minimum zoom level, and the view is centered on the first node. Click **Zoom In** to view the image at a larger size, and click **Zoom Out** to view the image (when the entire image is not displayed) at a smaller size. The **Zoom** command allows you to specify a magnification percentage (100% is considered the normal size for the schematic symbols).

Within the schematic view, you can also use the following mouse gestures to zoom in on a specific section:

- **zoom in**—Dragging a box around an area starting in the upper-left and dragging to the lower right zooms in on that area.
- **zoom -0.5**—Dragging a line from lower-left to upper-right zooms out 0.5 levels of magnification.
- **zoom 0.5**—Dragging a line from lower-right to upper-left zooms in 0.5 levels of magnification.
- **zoom fit**—Dragging a line from upper-right to lower-left fits the schematic view in the page.

You can also use the Zoom Tool on the Netlist Viewer toolbar to control magnification in the schematic view. When you select the Zoom Tool in the toolbar, clicking in the schematic zooms in and centers the view on the location you clicked. Right-click in the schematic to zoom out and center the view on the location you clicked. When you select the Zoom Tool, you can also zoom into a certain portion of the schematic by selecting a rectangular box area with your mouse cursor. The schematic is enlarged to show the selected area.

Related Information

Filtering in the Schematic View on page 19-16

Navigating with the Bird's Eye View

To open the Bird’s Eye View, on the View menu, click **Bird’s Eye View**, or click on the **Bird’s Eye View** icon in the toolbar.

Viewing the entire schematic can be useful when debugging and tracing through a large netlist. The Quartus II software allows you to quickly navigate to a specific section of the schematic using the Bird’s Eye View feature, which is available in the RTL Viewer and Technology Map Viewer.

The Bird’s Eye View shows the current area of interest.

- Select an area by clicking and dragging the indicator or right-clicking to form a rectangular box around an area.
- Click and drag the rectangular box to move around the schematic.
- Resize the rectangular box to zoom-in or zoom-out in the schematic view.

Partition the Schematic into Pages

For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view.

When a hierarchy level is partitioned into multiple pages, the title bar for the schematic window indicates which page is displayed and how many total pages exist for this level of hierarchy. The schematic view displays this as **Page** <*current page number*> **of** <*total number of pages*>.

Related Information

Introduction to the User Interface on page 19-5

Follow Nets Across Schematic Pages

Input and output connector symbols indicate nodes that connect across pages of the same hierarchy. Double-click a connector to trace the net to the next page of the hierarchy.

Note: After you double-click to follow a connector port, the Netlist Viewer opens a new page, which centers the view on the particular source or destination net using the same zoom factor as the previous page. To trace a specific net to the new page of the hierarchy, Altera recommends that you first select the necessary net, which highlights it in red, before you double-click to traverse pages.

Related Information

Schematic Symbols on page 19-11

State Machine Viewer

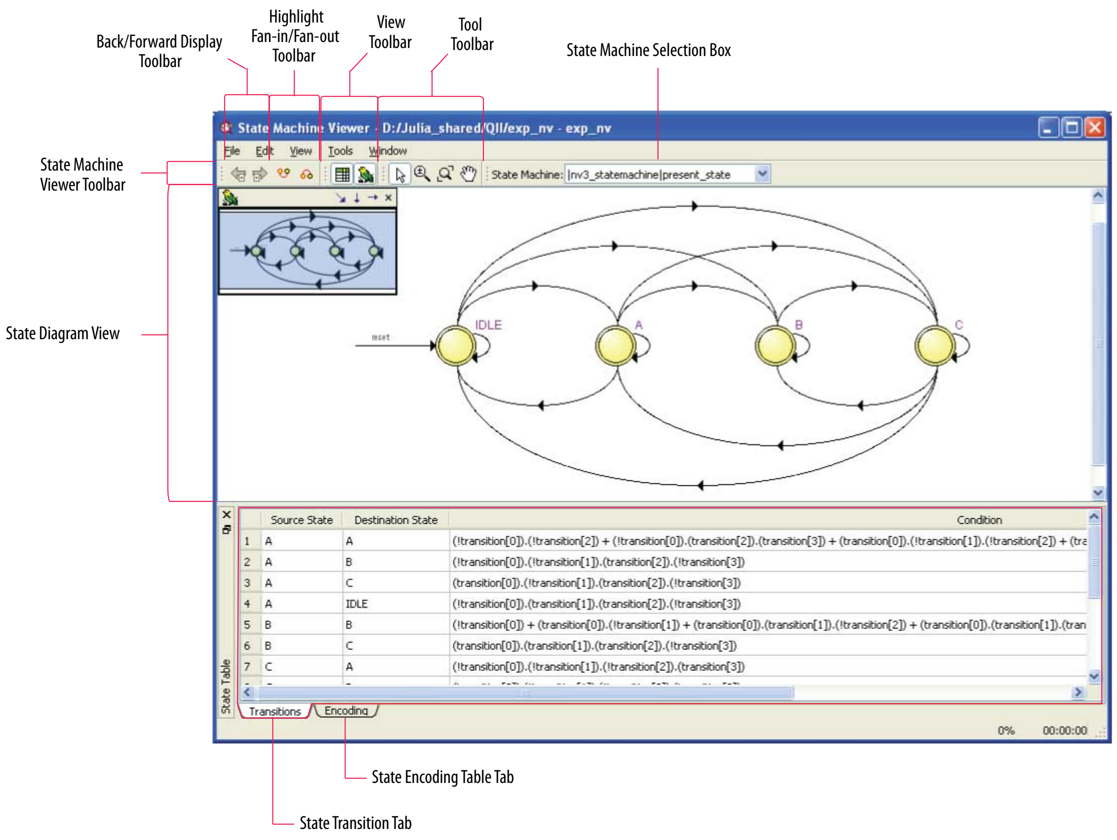
The State Machine Viewer displays a graphical representation of the state machines in your design.

You can open the State Machine Viewer in any of the following ways:

- On the Tools menu, point to **Netlist Viewers** and click **State Machine Viewer**.
- Double-click a state machine instance in the RTL Viewer

Figure 19-7: The State Machine Viewer

The following figure shows an example of the State Machine Viewer for a simple state machine and lists the components of the viewer.


Back/Forward Display Toolbar
Highlight Fan-in/Fan-out Toolbar
View Toolbar
Tool Toolbar
State Machine Selection Box
State Machine Viewer Toolbar
State Diagram View
State Machine Viewer - D:/Julia_shared/QII/exp_nv - exp_nv
File Edit View Tools Window
State Machine: |nv3_statemachine|present_state
reset
IDLE
A
B
C
State Table
Transitions
Encoding
0%
00:00:00
State Encoding Table Tab
State Transition Tab

	Source State	Destination State	Condition
1	A	A	(!transition[0]).(!transition[2]) + (!transition[0]).(transition[2]).(transition[3]) + (transition[0]).(!transition[1]).(!transition[2]) + (tra
2	A	B	(!transition[0]).(!transition[1]).(transition[2]).(!transition[3])
3	A	C	(transition[0]).(!transition[1]).(transition[2]).(!transition[3])
4	A	IDLE	(!transition[0]).(transition[1]).(transition[2]).(!transition[3])
5	B	B	(!transition[0]) + (transition[0]).(!transition[1]) + (transition[0]).(transition[1]).(!transition[2]) + (transition[0]).(transition[1]).(tran
6	B	C	(transition[0]).(transition[1]).(transition[2]).(!transition[3])
7	C	A	(!transition[0]).(!transition[1]).(!transition[2]).(transition[3])

State Diagram View

The state diagram view appears at the top of the State Machine Viewer. It contains a diagram of the states and state transitions.

The nodes that represent each state are arranged horizontally in the state diagram view with the initial state (the state node that receives the reset signal) in the left-most position. Nodes that connect to logic outside of the state machine instance are represented by a double circle. The state transition is represented by an arc with an arrow pointing in the direction of the transition.

When you select a node in the state diagram view, and turn on the **Highlight Fan-in** or **Highlight Fan-out** command from the View menu or the State Machine Viewer toolbar, the respective fan-in or fan-out transitions from the node are highlighted in red.

Note: An encrypted block with a state machine displays encoding information in the state encoding table, but does not display a state transition diagram or table.

State Transition Table

The state transition table on the **Transitions** tab at the bottom of the State Machine Viewer displays the condition equation for each state transition.

Each row in the table represents a transition (each arc in the state diagram view). The table has the following columns:

- **Source State**—the name of the source state for the transition
- **Destination State**—the name of the destination state for the transition
- **Condition**—the condition equation that causes the transition from source state to destination state

To see all of the transitions to and from each state name, click the appropriate column heading to sort on that column.

The text in each column is left-aligned by default; to change the alignment and to make it easier to see the relevant part of the text, right-click the column and click **Align Right**. To revert to left alignment, click **Align Left**.

Click in any cell in the table to select it. To select all cells, right-click in the cell and click **Select All**; or, on the Edit menu, click **Select All**. To copy selected cells to the clipboard, right-click the cells and click **Copy Table**; or, on the Edit menu, point to **Copy** and click **Copy Table**. You can paste the table into any text editor as tab-separated columns.

State Encoding Table

The state encoding table on the **Encoding** tab at the bottom of the State Machine Viewer displays encoding information for each state transition.

To view state encoding information in the State Machine Viewer, you must synthesize your design with the **Start Analysis & Synthesis** command. If you have only elaborated your design with the **Start Analysis & Elaboration** command, the encoding information is not displayed.

Select Items in the State Machine Viewer

You can select and highlight each state node and transition in the State Machine Viewer. To select a state transition, click the arc that represents the transition.

When you select a node or transition arc in the state diagram view, the matching state node or equation conditions in the state transition table are highlighted; conversely, when you select a state node or equation condition in the state transition table, the corresponding state node or transition arc is highlighted in the state diagram view.

Switch Between State Machines

A design may contain multiple state machines. To choose which state machine to view, use the **State Machine** selection box located at the top of the State Machine Viewer. Click in the drop-down box and select the necessary state machine.

Probing to a Source Design File and Other Quartus II Windows

The RTL Viewer, Technology Map Viewer, and State Machine Viewer allow you to cross-probe to the source design file and to various other windows in the Quartus II software.

You can select one or more hierarchy boxes, nodes, state nodes, or state transition arcs that interest you in the Netlist Viewer and locate the corresponding items in another applicable Quartus II software window. You can then view and make changes or assignments in the appropriate editor or floorplan.

To locate an item from the Netlist Viewer in another window, right-click the items of interest in the schematic or state diagram, point to **Locate**, and click the appropriate command. The following commands are available:

- **Locate in Assignment Editor**
- **Locate in Pin Planner**
- **Locate in Chip Planner**
- **Locate in Resource Property Editor**
- **Locate in Technology Map Viewer**
- **Locate in RTL Viewer**
- **Locate in Design File**

The options available for locating an item depend on the type of node and whether it exists after placement and routing. If a command is enabled in the menu, it is available for the selected node. You can use the **Locate in Assignment Editor** command for all nodes, but assignments might be ignored during placement and routing if they are applied to nodes that do not exist after synthesis.

The Netlist Viewer automatically opens another window for the appropriate editor or floorplan and highlights the selected node or net in the newly opened window. You can switch back to the Netlist Viewer by selecting it in the Window menu or by closing, minimizing, or moving the new window.

Probing to the Netlist Viewers from Other Quartus II Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows in the Quartus II software. You can select one or more nodes or nets in another window and locate them in one of the Netlist Viewers.

You can locate nodes between the RTL Viewer, State Machine Viewer, and Technology Map Viewer, and you can locate nodes in the RTL Viewer and Technology Map Viewer from the following Quartus II software windows:

- Project Navigator
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Node Finder

- Assignment Editor
- Messages Window
- Compilation Report
- TimeQuest Timing Analyzer (supports the Technology Map Viewer only)

To locate elements in the Netlist Viewer from another Quartus II window, select the node or nodes in the appropriate window; for example, select an entity in the **Entity** list on the **Hierarchy** tab in the Project Navigator, or select nodes in the Timing Closure Floorplan, or select node names in the **From** or **To** column in the Assignment Editor. Next, right-click the selected object, point to **Locate**, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. After you click this command, the Netlist Viewer opens, or is brought to the foreground if the Netlist Viewer is open.

When cross-probing from the Time

Note: The first time the window opens after a compilation, the preprocessor stage runs before the Netlist Viewer opens.

The Netlist Viewer shows the selected nodes and, if applicable, the connections between the nodes. The display is similar to what you see if you right-click the object, point to **Filter**, and click **Selected Nodes** using **Filter across hierarchy**. If the nodes cannot be found in the Netlist Viewer, a message box displays the message: **Can't find requested location**.

Viewing a Timing Path

You can cross-probe from a report panel in the TimeQuest Timing Analyzer to see a visual representation of a timing path.

To take advantage of this feature, you must complete a full compilation of your design, including the timing analyzer stage. To see the timing results for your design, on the Processing menu, click **Compilation Report**. On the left side of the Compilation Report, select **TimeQuest Timing Analyzer**. When you select a detailed report, the timing information is listed in a table format on the right side of the Compilation Report; each row of the table represents a timing path in the design. You can also view timing paths in TimeQuest analyzer report panels. To view a particular timing path in the Technology Map Viewer or RTL Viewer, right-click the appropriate row in the table, point to **Locate**, and click **Locate in Technology Map Viewer** or **Locate in RTL Viewer**.

- To locate a path, on the **Tasks** pane, in the **Custom Reports** folder, double-click **Report Timing**.
- In the **Report Timing** dialog box, make necessary settings, and then click the **Report Timing** button.
- After the TimeQuest analyzer generates the report, right-click on the node in the table and select **Locate Path**. In the Technology Map Viewer, the schematic page displays the nodes along the timing path with a summary of the total delay.

When you locate the timing path from the TimeQuest analyzer to the Technology Map Viewer, the interconnect and cell delay associated with each node is displayed on top of the schematic symbols. The total slack of the selected timing path is displayed in the Page Title section of the schematic.

In the RTL Viewer, the schematic page displays the nodes in the paths between the source and destination registers with a summary of the total delay.

The RTL Viewer netlist is based on an initial stage of synthesis, so the post-fitting nodes might not exist in the RTL Viewer netlist. Therefore, the internal delay numbers are not displayed in the RTL Viewer as they are in the Technology Map Viewer, and the timing path might not be displayed exactly as it appears in the timing analysis report. If multiple paths exist between the source and destination registers, the RTL Viewer might display more than just the timing path. There are also some cases in which the path cannot be displayed, such as paths through state machines, encrypted intellectual property (IP), or registers that are created during the fitting process. In cases where the timing path displayed in the RTL Viewer might not be the correct path, the compiler issues messages.

Document Revision History

Date	Version	Changes
2014.06.30	14.0.0	Added Show Netlist on One Page and show/Hide Instance Pins commands.
November 2013	13.1.0	Removed HardCopy device information. Reorganized and migrated to new template. Added support for new Netlist viewer.
November 2012	12.1.0	Added sections to support Global Net Routing feature.
June 2012	12.0.0	Removed survey link.
November 2011	10.0.2	Template update.
December 2010	10.0.1	Changed to new document template.
July 2010	10.0.0	• Updated screenshots • Updated chapter for the Quartus II software version 10.0, including major user interface changes
November 2009	9.1.0	• Updated devices • Minor text edits

Date	Version	Changes
March 2009	9.0.0	• Chapter 13 was formerly Chapter 12 in version 8.1.0 • Updated Figure 13–2, Figure 13–3, Figure 13–4, Figure 13–14, and Figure 13–30 • Added "Enable or Disable the Auto Hierarchy List" on page 13–15 • Updated "Find Command" on page 13–44
November 2008	8.1.0	Changed page size to 8.5" × 11"

Date	Version	Changes
May 2008	8.0.0	• Added Arria GX support • Updated operator symbols • Updated information about the radial menu feature • Updated zooming feature • Updated information about probing from schematic to SignalTap II Analyzer • Updated constant signal information • Added .png and .gif to the list of supported image file formats • Updated several figures and tables • Added new sections "Enabling and Disabling the Radial Menu", "Changing the Time Interval", "Changing the Constant Signal Value Formatting", "Logic Clouds in the RTL Viewer", "Logic Clouds in the Technology Map Viewer", "Manually Group and Ungroup Logic Clouds", "Customizing the Shortcut Commands" • Renamed several sections • Removed section "Customizing the Radial Menu" • Moved section "Grouping Combinational Logic into Logic Clouds" • Updated document content based on the Quartus II software version 8.0

Related Information

Quartus II Handbook Archive

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