Quartus Prime Pro Edition Handbook Volume 1: Design and Synthesis

Subscribe

Send Feedback

QPP5V1
2015.11.02

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

Introduction to Quartus Prime Pro Edition 1

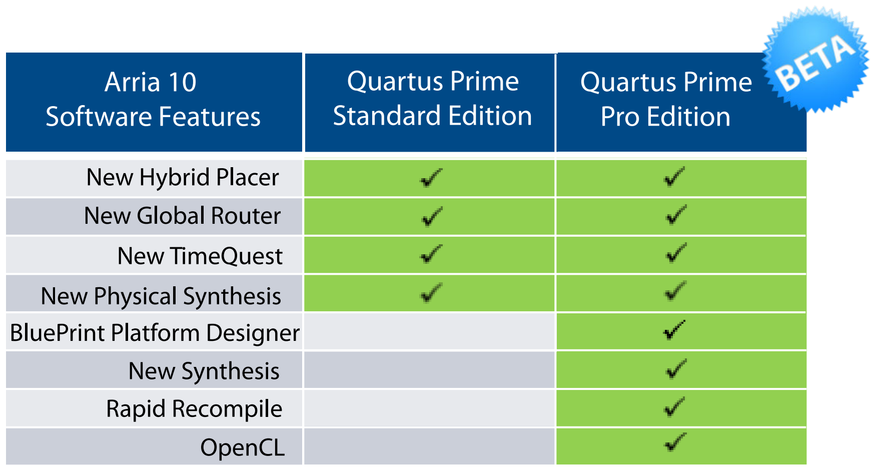
2015.11.02

QPP5V1 Subscribe Send Feedback

The Quartus® Prime software provides a complete design environment for CPLD, FPGA, and SoC designs. The user interface supports easy design entry, fast processing, and straightforward device programming. The Quartus Prime Pro Edition software adds the Spectra-Q engine, which enables next generation synthesis, physical optimization, design methodologies, and FPGA architectures.

The Spectra-Q engine is optimized for Arria 10 and future devices, and provides powerful and customizable design processing to achieve the best possible silicon implementation of your project. The Quartus Prime software makes it easy for you to focus on your design—not on the tool. Only the Quartus Prime Pro Edition software includes all of the unique features of the Spectra-Q engine.

Figure 1-1: Quartus Prime New Feature Support Matrix for 15.1 Beta



Arria 10 Software Features	Quartus Prime Standard Edition	Quartus Prime Pro Edition
New Hybrid Placer	✓	✓
New Global Router	✓	✓
New TimeQuest	✓	✓
New Physical Synthesis	✓	✓
BluePrint Platform Designer		✓
New Synthesis		✓
Rapid Recompile		✓
OpenCL		✓

The Spectra-Q engine helps to streamline the FPGA development process and ensure highest performance for the least effort. Quartus Prime Pro Edition software 15.1 beta supports Arria 10 devices and includes the following enhancements:

- Hierarchical project database—a separate compilation database preserves individual post-synthesis, post-placement, and post-place-and-route results for each partition
- New Spectra-Q synthesis—stricter language parser with enhanced language support, faster algorithms, and true parallel synthesis
- New Spectra-Q physical synthesis optimization—performs combinational and sequential optimization during fitting to improve circuit performance
- Faster more accurate I/O placement—plan interface I/O in BluePrint Platform Designer
- Rapid Recompile—accelerates compilation by reusing verified results
- Improved routing preservation—SignalTap II Logic Analyzer implementation now includes routing preservation

Related Information

- **Migrating to Quartus Prime Pro Edition** on page 1-3
- **Upgrade Project Assignments and Constraints** on page 1-3
- **Upgrade IP Cores and Qsys Systems** on page 1-8
- **Upgrade Non-Compliant Design RTL** on page 1-9

Should I Choose the Quartus Prime Pro Edition Software?

Depending on your immediate needs, the Quartus Prime Pro Edition software may be an appropriate choice for your design.

In version 15.1, the Quartus Prime Pro Edition software is a public beta, with known limitations that may be important to you. Consider these limitations, together with the advantages of the Spectra-Q engine, in informing your decision.

Quartus Prime Pro Edition 15.1 Beta Limitations

Quartus Prime Pro Edition software for 15.1 beta release provides no support for the following features:

- Back annotation
- OpenCore Plus time-limited IP evaluation
- NativeLink third party tool integration
- Video and Image Processing Suite IP Cores in Qsys
- Export of version-compatible databases

Quartus Prime Pro Edition Decision Points

If these features are not essential, consider the needs and timelines of your project in determining whether the Quartus Prime Standard Edition or Quartus Prime Pro Edition software is most appropriate for you.

The following decision tree may help you decide whether to use the Quartus Prime Pro Edition beta software.

- If you are not using Arria 10, the Quartus Prime Standard Edition software meets your needs.
- If you have an existing Arria 10 design which doesn't require the additional features of the Quartus Prime Pro Edition, the Quartus Prime Standard Edition software meets your needs..
- If you are beginning a new Arria 10 design, or have an existing Arria 10 design that would benefit from the new features enabled by the Spectra-Q engine, then the Quartus Prime Pro Edition software is for you.

Related Information

- **Managing Quartus Prime Projects** on page 2-1
- **Design Compilation** on page 12-1

Migrating to Quartus Prime Pro Edition

The Quartus Prime Pro Edition software supports migration of Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software projects.

Note: The migration steps for Quartus Prime Lite Edition, Quartus Prime Standard Edition, and the Quartus II software are identical. For brevity, this section refers to these Altera tools collectively as "other Quartus software products."

Migrating to Quartus Prime Pro Edition requires the following changes to other Quartus software product projects:

1. Upgrade project assignments and constraints with equivalent Quartus Prime Pro Edition assignments
2. Upgrade all Altera IP core variations and Qsys systems in your project
3. Upgrade design RTL to standards-compliant VHDL, Verilog HDL, or SystemVerilog

This document describes each migration step in detail.

Keep Pro Edition Project Files Separate

Quartus Prime Pro Edition software does not support project or constraint files from other Quartus software products. Do not place project files from other Quartus software products within in the same directory as Quartus Prime Pro Edition project files. In general, use Quartus Prime Pro Edition project files and directories only for Quartus Prime Pro Edition projects, and use other Quartus software product files only with those software tools.

Quartus Prime Pro Edition projects do not support compilation in other Quartus software products, and vice versa. The Quartus Prime Pro Edition software generates an error if it detects other Quartus software product features in project files.

Before migrating other Quartus software product projects, click **Project** > **Archive Project** to save a copy of your original project before making modifications for migration.

Upgrade Project Assignments and Constraints

Quartus Prime Pro Edition software introduces changes to handling of project assignments and constraints that the Quartus Settings File (**.qsf**) stores. You must upgrade other Quartus software product project assignments and constraints for migration to the Quartus Prime Pro Edition software. Upgrade other Quartus software product assignments with **Assignments** > **Assignment Editor**, by editing the **.qsf** file directly, or by using a Tcl script.

The following sections detail each type project assignment upgrade that migration requires.

Related Information

- **Modify Entity Name Assignments** on page 1-4
- **Resolve SDC Entity Names** on page 1-4
- **Verify Generated Node Name Assignments** on page 1-5
- **Replace LogicLock Regions** on page 1-5
- **Modify SignalTap II Logic Analyzer Files** on page 1-8
- **Remove Unsupported Feature Assignments** on page 1-8

Modify Entity Name Assignments

Quartus Prime Pro Edition software supports assignments that include instance names *without* a corresponding entity name, as shown in the following example:

- "`a_entity:a|b_entity:b|c_entity:c`" (includes deprecated entity names)
- “`a|b|c`” (omits deprecated instance names)

While the current version of the Quartus Prime Pro Edition software still *accepts* entity names in the **.qsf**, the Compiler *ignores* the entity name. The Compiler generates a warning message if it detects entity names in the **.qsf**. Whenever possible, you should remove entity names from assignments, and discontinue reliance on entity-based assignments. Future versions of the Quartus Prime Pro Edition software may eliminate all support for entity-based assignments.

Resolve SDC Entity Names

The Quartus Prime Pro Edition TimeQuest timing analyzer honors entity names in Synopsys Design Constraints (**.sdc**) files, and generates no associated warnings. You can generally use **.sdc** files from other Quartus software products without modification.

However, you may need to modify any scripts that include custom processing of names that the SDC command returns, such as `get_registers`. Your script must reflect that returned strings do not include entity names.

The SDC commands respect wildcard patterns containing entity names. Review the TimeQuest reports to verify application of all constraints. The following example illustrates differences between functioning and non-functioning SDC scripts:

```
# Apply a constraint to all registers named "acc" in the entity "counter".
# This constraint functions in both SE and PE, because the SDC
# command always understands wildcard patterns with entity names in them
set_false_path –to [get_registers  “counter:*|*acc”]

# This does the same thing, but first it converts all register names to
# strings, which includes entity names by default in the SE
# but excludes them by default in the PE. The regexp will therefore
# fail in PE by default.
#
# This script would also fail in the SE, and earlier
# versions of Quartus II, if entity name display had been disabled
# in the QSF.
set all_reg_strs [query_collection –list –all [get_registers *]]
foreach keeper $all_reg_strs {
   if {[regexp {counter:*|:*acc} $keeper]} {
      set_false_path –to $keeper
   }
}
```

In some cases, you cannot remove the entity name processing from **.sdc** files due to complex processing involving node names. Use standard SDC whenever possible to replace such processing. Alternatively, add the following code to the top and bottom of your script to temporarily re-enable entity name display in the **.sdc** file:

```
# This script requires that entity names be included
# due to custom name processing
set old_mode [set_project_mode -get_mode_value always_show_entity_name]
set_project_mode -always_show_entity_name on

<... the rest of your script goes here ...>

# Restore the project mode
set_project_mode -always_show_entity_name $old_mode
```

Verify Generated Node Name Assignments

Quartus Prime synthesis generates and automatically names internal design nodes during processing. The Quartus Prime Pro Edition uses different conventions than other Quartus software products[1] to generate node names during synthesis. When you synthesize your other Quartus software product project in Quartus Prime Pro Edition, the synthesis-generated node names may change. If any scripts or constraints depend on the synthesis-generated node names, you must update the script or constraint to match the Quartus Prime Pro Edition synthesis node names.

Avoid dependence on synthesis-generated names due to frequent changes in name generation. In addition, verify the names of duplicated registers and PLL clock output to ensure compatibility with any script or constraint.

Replace LogicLock Regions

Quartus Prime Pro Edition software introduces more simplified and flexible LogicLock® constraints, compared with other Quartus software product LogicLock regions. You must replace all LogicLock assignments with compatible LogicLock Plus assignments for migration. To convert LogicLock regions to LogicLock Plus regions:

1. remove all existing LogicLock assignments using one of the following methods:
   - **Assignments** > **Assignment Editor**
   - **Assignments** > **Regions Window**
   - Edit the **.qsf**
2. Edit the **.qsf** or use **Assignments** > **Chip Planner** to create new region constraints that match the functionality of your original project.

Compilation fails if synthesis finds other Quartus software product LogicLock assignments the Quartus Prime Pro Edition project. The following table compares other Quartus software product region constraint support with the Quartus Prime Pro Edition software.

[1] For brevity, this section refers to Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software collectively as "other Quartus software products."

Table 1-1: Region Constraints Per Edition

Constraint Type	LogicLock Region Support Other Quartus Software Products	LogicLock Plus Support Quartus Prime Pro Edition
Fixed rectangular, nonrectangular or non-contiguous regions	Full support.	Full support.
Chip Planner entry	Full support.	Full support.
Periphery element assignments	Supported in some instances.	Full support. Use “core-only” regions to exclude the periphery.
Nested (“hierarchical”) regions	Supported but separate hierarchy from the user instance tree.	Supported in same hierarchy as user instance tree.
Reserved regions	Limited support for nested or nonrectangular reserved regions. Reserved regions typically cannot cross I/O columns; non-contiguous regions must be used instead.	Full support for nested and nonrectangular regions. Reserved regions can cross I/O columns without affecting periphery logic if they are "core-only".
Routing regions	Limited support via “routing expansion.” No support with hierarchical regions.	Full support (including future support for hierarchical regions).
Floating or autosized regions	Full support.	No support.
Region names	Regions have names.	Regions are identified by the instance name of the constrained logic.
Multiple instances in the same region	Full support.	Full support for non-reserved regions; multiple instances may be assigned to the same area. Not supported for reserved regions in this release.
Member exclusion	Full support.	No support for arbitrary logic. Use a core-only region to exclude periphery elements. Use non-rectangular regions to include more RAM or DSP columns as needed.

LogicLock Plus Region Assignment Examples

The following examples show the syntax for various LogicLock Plus region assignments in the **.qsf**.

You can also enter these assignments in the Assignment Editor, the Regions window, or the Chip Planner.

Example 1-1: Assign Rectangular LogicLock Plus Region

Assigns a rectangular LogicLock Plus region to a lower right corner location of (10,10), and an upper right corner of (20,20) inclusive.

```
set_instance_assignment -name PLACE_REGION -to a|b|c "10 10 20 20"
```

Example 1-2: Assign Non-Rectangular LogicLock Plus Region

Assigns instance "a|b|c" to non-rectangular L-shaped LogicLock Plus region. The software treats each set of four numbers as a new box.

```
set_instance_assignment -name PLACE_REGION -to x|y|z "10 10 20 50; 20 10 50
20"
```

Example 1-3: Assign Subordinate LogicLock Plus Instances

By default, the Quartus Prime software constrains every child instance to the LogicLock Plus region of its parent. Any constraint to a child instance intersects with the constraint of its ancestors. For example, in the following example, all logic beneath “a|b|c|d” constrains to box (10,10),(15,15), and not (0,0),(15,15). This result occurs because the child constraint intersects with the parent constraint.

```
set_instance_assignment -name PLACE_REGION -to a|b|c "10 10 20 20"
set_instance_assignment -name PLACE_REGION -to a|b|c|d "0 0 15 15"
```

Example 1-4: Assign Multiple LogicLock Plus Instances

By default, a LogicLock Plus region constraint allows logic from other instances to share the same region. In other words, the following assignments are in conflict.

```
set_instance_assignment -name PLACE_REGION -to a|b|c "10 10 20 20"
set_instance_assignment -name PLACE_REGION -to e|f|g "10 10 20 20"
```

Example 1-5: Assigned Reserved LogicLock Plus Regions

You can reserve an entire LogicLock Plus region for one instance and any of its subordinate instances.

```
set_instance_assignment -name PLACE_REGION -to a|b|c "10 10 20 20"
set_instance_assignment -name RESERVE_PLACE_REGION -to a|b|c ON

# The following assignment causes an error. The logic in e|f|g is not
# legally placeable anywhere:
# set_instance_assignment -name PLACE_REGION -to e|f|g "10 10 20 20"

# The following assignment does *not* cause an error, but is effectively
# constrained to the box (20,10),(30,20), since the (10,10),(20,20) box is
reserved
```

```
# for a|b|c
set_instance_assignment -name PLACE_REGION -to e|f|g "10 10 30 20"
```

Modify SignalTap II Logic Analyzer Files

Quartus Prime Pro Edition introduces new methodology for entity names, settings, and assignments. These changes impact the processing of SignalTap II Logic Analyzer Files (**.stp**). If you migrate a project that includes **.stp** files generated by other Quartus software products, you must make the following changes to migrate to the Quartus Prime Pro Edition:

1. Remove entity names from **.stp** files. The SignalTap II Logic Analyzer allows without error, but ignores, entity names in **.stp** files. Remove entity names from **.stp** files for migration to Quartus Prime Pro Edition:
   a. Click **View** > **Utility Windows** > **Node Finder** to quickly locate and remove appropriate nodes. Use Node Finder options to filter on nodes.
   b. Click **Processing** > **Start** > **Start Analysis & Elaboration** to repopulate the database and add valid node names.
2. Remove Post-Fit Nodes. Quartus Prime Pro Edition uses a different post-fit node naming scheme than other Quartus software products.
   a. Remove post-fit tap node names originating from other Quartus software products.
   b. Click **View** > **Utility Windows** > **Node Finder** to quickly locate and remove post-fit nodes. Use Node Finder options to filter on nodes.
   c. Click **Processing** > **Start** > **Start Compilation** to repopulate the database and add valid post-fit nodes.
3. Run Quartus Prime Pro Edition an initial compilation from the GUI. The Compiler automatically removes SignalTap II assignments originating other Quartus software products. Alternatively, from the command-line, run `quartus_stp` once on the project to remove outmoded assignments.

   Note: `quartus_stp` introduces no migration impact in the Quartus Prime Pro Edition. Your scripts require no changes to `quartus_stp` for migration.
4. Modify SDC Constraints for JTAG. Quartus Prime Pro Edition does not support embedded SDC constraints for JTAG signals. Modify the timing template to suit the design's JTAG driver (e.g. USB Blaster II) and board.

Remove Unsupported Feature Assignments

The Quartus Prime Pro Edition software does not support some feature assignments that other Quartus software products support. Remove the following unsupported feature assignments from other Quartus software product **.qsf** files for migration to the Quartus Prime Pro Edition software.

- Incremental Compilation (partitions)—The current version of the Quartus Prime Pro Edition software does not support incremental compilation. Remove all incremental compilation feature assignments from other Quartus software product **.qsf** files before migration. The Spectra-Q engine contains a new implementation of hierarchical compilation to be enabled in a subsequent release.
- HyperRetimer—The current version of the Quartus Prime Pro Edition software does not support Stratix 10 devices or this related feature. Remove all HyperRetimer feature assignments from other Quartus software product **.qsf** files before migration.

Upgrade IP Cores and Qsys Systems

Quartus Prime Pro Edition uses standards-compliant methodology for instantiation and generation of IP cores and Qsys systems. You must upgrade all IP cores and Qsys systems in your project for migration of other Quartus software product projects to the Quartus Prime Pro Edition software. Most Altera IP cores and Qsys systems upgrade automatically in the **Upgrade IP Components** dialog box.

Other Quartus software products[2] use a proprietary Verilog configuration scheme within the top level of IP cores and Qsys systems for synthesis file. The Quartus Prime Pro Edition does not support this scheme. To upgrade all IP cores and Qsys systems in your project, click **Project** > **Upgrade IP Components**.

Table 1-2: IP Core and Qsys System Differences

Other Quartus Software Products	Quartus Prime Pro Edition
IP and Qsys system generation use a proprietary Verilog configuration scheme within the top level of IP cores and Qsys systems for synthesis files. This proprietary Verilog configuration scheme prevents RTL entities from ambiguous instantiation errors during synthesis. However, these errors may manifest in simulation. Resolving this issue requires writing a Verilog configuration to disambiguate the instantiation, delete the duplicate entity from the project, or rename one of the conflicting entities. Quartus Prime Pro Edition IP strategy resolves these issues.	IP and Qsys system generation does not use proprietary Verilog configurations. The compilation library scheme changes in the following ways: • Compiles all variants of an IP core into the same compilation library across the entire project. Quartus Prime Pro Edition identically names IP cores with identical functionality and parameterization to avoid ambiguous entity instantiation errors. For example, the files for every Arria 10 PCI Express IP core variant compile into the **altera_pcie_a10_hip_151** compilation library. • Simulation and synthesis file sets for IP cores and systems instantiate entities in the same manner. • The generated RTL directory structure now matches the compilation library structure.

Note: For complete information on upgrading IP cores and Qsys systems, refer to Managing Projects in the *Quartus Prime Handbook*.

Related Information

- **Introduction to Altera IP Cores**
- **Upgrading IP Cores**
- **Managing Quartus Prime Projects** on page 2-1

Upgrade Non-Compliant Design RTL

The Quartus Prime Pro Edition software introduces a new Sprectra-Q synthesis engine (`quartus_syn` executable). The Spectra-Q synthesis engine enforces stricter industry-standard HDL structures and supports the following enhancements in this release:

- More robust support for SystemVerilog
- Improved support for VHDL2008
- New RAM inference engine infers RAMs from GENERATE statements or array of integers
- Stricter syntax/semantics check for improved compatibility with other EDA tools

[2] For brevity, this section refers to Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software collectively as "other Quartus software products."

You must account for these synthesis differences in existing RTL code by ensuring that your design uses standards-compliant VHDL, Verilog HDL, or SystemVerilog. The Compiler generates errors when processing non-compliant RTL. Spectra-Q engine technology also provides foundational support for future Quartus Prime Pro Edition software enhancements. Use the guidelines in this section to modify existing RTL for compatibility with Spectra-Q synthesis.

Related Information

- **Verify Verilog Compilation Unit** on page 1-10
- **Update Entity Auto Discovery** on page 1-11
- **Ensure Distinct VHDL Namespace for Each Library** on page 1-11
- **Remove Unsupported Parameter Passing** on page 1-12
- **Remove Filling Vectors from WYSIWYG Instantiation** on page 1-12
- **Remove Non-Standard Pragmas** on page 1-12
- **Declare Objects Before Initial Values** on page 1-13
- **Confine SystemVerilog Features to SystemVerilog Files** on page 1-13
- **Avoid Assignment Mixing in Always Blocks** on page 1-14
- **Avoid Unconnected, Non-Existent Ports** on page 1-14
- **Avoid Illegal Parameter Ranges** on page 1-14
- **Update Verilog and VHDL Type Mapping** on page 1-14

Verify Verilog Compilation Unit

The Verilog LRM defines the concept of compilation unit as “a collection of one or more Verilog source files compiled together” forming the compilation-unit scope. Items visible only in the compilation-unit scope include macros, global declarations, and default net types. The contents of included files become part of the compilation unit of the parent file. Modules, primitives, programs, interfaces, and packages are visible in all compilation units. Quartus Prime Pro Edition synthesis uses a different method to define the compilation unit. Ensure that your RTL accommodates these changes.

Table 1-3: Verilog Compilation Unit Differences

Other Quartus Software Products	Quartus Prime Pro Edition
Synthesis in other Quartus software products follows the Multi-file compilation unit (MFCU) method to select compilation unit files. In MFCU, all files compile in the same compilation unit. Global definitions and directives are visible in all files. However, the default net type is reset at the start of each file.	Quartus Prime Pro Edition synthesis follows the Single-file compilation unit (SFCU) method to select compilation unit files. In SFCU, each file is a compilation unit. For example, a macro declared in the first file would remain defined in all subsequent files unless undefined. File order is thus important.

Note: You can optionally change to MFCU mode using the following assignment:

```
set_global_assignment -name VERILOG_CU_MODE MFCU
```

Verilog Configuration Instantiation

In other Quartus software products, synthesis automatically finds any Verilog configuration relating to a module that you instantiate. Quartus Prime Pro Edition synthesis requires instantiation of the Verilog configuration, and not the module. The Verilog configuration then in turn instantiates the design.

If your top-level entity is a Verilog configuration, set the Verilog configuration, rather than the module, as the top-level entity.

Other Quartus Software Products	Quartus Prime Pro Edition
From the Example RTL, synthesis automatically finds the `mid_config` Verilog configuration relating to the instantiated module.	From the Example RTL, synthesis does not find the `mid_config` Verilog configuration. You must instantiate the Verilog configuration directly.

Example RTL:

```
config mid_config;
design good_lib.mid;
instance mid.sub_inst use good_lib.sub;
endconfig

module test (input a1, output b);
mid_config mid_inst ( .a1(a1), .b(b));
// in other Quartus products preceding line would have been:
//mid mid_inst ( .a1(a1), .b(b));
endmodule

module mid (input a1, output b);
sub sub_inst (.a1(a1), .b(b));
endmodule
```

Update Entity Auto Discovery

All editions of the Quartus Prime and Quartus II software search your project directory for undefined entities. For example, if you instantiate entity "sub" in your design without specifying "sub" as a design file in the Quartus Settings File (**.qsf**) , synthesis searches for **sub.v, sub.vhd**, and so on. However, Quartus Prime Pro Edition performs auto-discovery at a different stage in the flow. Ensure that your RTL code accommodates these auto discovery changes.

Table 1-4: Entity Auto Discovery Differences

Other Quartus Software Products	Quartus Prime Pro Edition
Always automatically searches your project directory and search path for undefined entities.	Always automatically searches your project directory and search path for undefined entities. Quartus Prime Pro Edition synthesis performs auto-discovery earlier in the flow than other Quartus software products. This results in discovery of more syntax errors. Optionally disable auto discovery with the following **.qsf** assignment: `set_global_assignment -name AUTO_DISCOVER_AND_SORT OFF`

Ensure Distinct VHDL Namespace for Each Library

Quartus Prime Pro Edition synthesis requires that VHDL namespaces are distinct for each library. The stricter library binding requirement complies with VHDL language specifications and results in deterministic behavior. This benefits team-based projects by avoiding unintentional name collisions. Confirm that your RTL respects this change.

Table 1-5: VHDL Namespace Differences

Other Quartus Software Products	Quartus Prime Pro Edition
For the Example RTL, the analyzer searches all libraries in an unspecified order until it finds package `utilities_pack` and uses items from that package. If another library, for example `projectLib` also contains `utilities_pack`, the analyzer may use this library instead of `myLib.utilites_pack` if found before the analyzer searches `myLib`.	For the Example RTL, the analyzer uses the specific `utilities_pack` in `myLib`. If `utilities_pack` does not exist in library `myLib`, the analyzer generates an error.

Example RTL:
 	
`library myLib; use`
`myLib.utilities_pack.all;`	

Remove Unsupported Parameter Passing

Synthesis in other Quartus software products supports passing parameters via the `set_parameter` command in the **.qsf**. Quartus Prime Pro Edition synthesis does not support this form of parameter passing in this version of the software. Ensure that your RTL does not depend on this type of parameter passing.

Table 1-6: SystemVerilog Feature Differences

Other Quartus Software Products	Quartus Prime Pro Edition
From the Example RTL, synthesis overwrites the value of parameter SIZE of instance of `my_ram` instantiated from entity `mid-level`.	From the Example RTL, synthesis generates a syntax error for detection of parameter passing assignments in the **.qsf**. Specify parameters in the RTL. The following example shows the supported top-level parameter passing format. This example applies only to the top-level and sets a value of 4 to parameter N: `set_parameter -name N 4`
Example RTL: `set_parameter -entity mid_level -to my_ram -name SIZE 16`	

Remove Filling Vectors from WYSIWYG Instantiation

Synthesis in other Quartus software products allows use of SystemVerilog (**.sv**) filling vectors when instantiating a WYSIWYG in a **.v** file. Quartus Prime Pro Edition synthesis does not allow use of filling vectors for WYSIWYG instantiation.

Quartus Prime Pro Edition synthesis allows use of filling vectors in **.sv** files for uses other than WYSIWYG instantiation. Ensure that your RTL code does not use filling vectors for WYSIWYG instantiation.

Remove Non-Standard Pragmas

Synthesis in other Quartus software products supports the `vhdl(verilog)_input_version` pragma and the `library` pragma. Quartus Prime Pro Edition synthesis does not support either of these pragmas.

Remove any use of the pragmas from RTL for Quartus Prime Pro Edition migration. Use the following guidelines to implement the pragma functionality in Quartus Prime Pro Edition:

- `vhdl(verilog)_input_version` Pragma—allows change to the input version in the middle of an input file. For example, to change VHDL 1993 to VHDL 2008. For Quartus Prime Pro Edition migration, specify the input version for each file in the **.qsf**.
- `library` Pragma—allows changes to the VHDL library into which files compile. For Quartus Prime Pro Edition migration, specify the compilation library in the **.qsf**.

Declare Objects Before Initial Values

Other Quartus software products allows declaration of initial value prior to declaration of the object. Quartus Prime Pro Edition synthesis requires declaration of objects before initial value. Ensure that your RTL declares objects before initial value.

Table 1-7: Object Declaration Differences

Other Quartus Software Products	Quartus Prime Pro Edition
From the Example RTL, synthesis initializes the output `p_prog_io1` with the value of `p_progio1_reg`, even though the register declaration occurs in Line 2.	From the Example RTL, synthesis generates a syntax error when you specify initial values before declaring the register.

Example RTL:
`1 output p_prog_io1 = p_prog_io1_reg;`
`2 reg p_prog_io1_reg;`	

Confine SystemVerilog Features to SystemVerilog Files

Other Quartus software products allow use of a subset of SystemVerilog (**.sv**) features in Verilog HDL (**.v**) design files. Quartus Prime Pro Edition synthesis does not allow SystemVerilog features in Verilog HDL files. To avoid syntax errors in Quartus Prime Pro Edition, allow only SystemVerilog features in Verilog HDL files.

To use SystemVerilog features in your existing Verilog files, rename your Verilog (**.v**) files as SystemVerilog (**.sv**) files.

Table 1-8: SystemVerilog Feature Differences

Other Quartus Software Products	Quartus Prime Pro Edition
From the Example RTL, synthesis interprets `$clog2` in a **.v** file, even though the Verilog LRM does not define the `$clog2` feature. Other Quartus software products allow other SystemVerilog features in **.v** files.	From the Example RTL, synthesis generates a syntax error for detection of any non-Verilog construct in **.v** files. Quartus Prime Pro Edition synthesis honors SystemVerilog features only in **.sv** files.

Example RTL:
`localparam num_mem_locations = 1050;`
`wire mem_addr [$clog2(num_mem_locations)-1 : 0];`	

Avoid Assignment Mixing in Always Blocks

Other Quartus software products allow mixed use of blocking and non-blocking assignments within `ALWAYS` blocks. Quartus Prime Pro Edition synthesis does not allow mixed use of blocking and non-blocking assignments within `ALWAYS` blocks. To avoid syntax errors, ensure that `ALWAYS` block assignments are of the same type for Quartus Prime Pro Edition migration.

Table 1-9: ALWAYS Block Assignment Differences

Other Quartus Software Products	Quartus Prime Pro Edition
Synthesis honors the mixed blocking and non-blocking assignments, although the Verilog Language Specification no longer supports this construct.	Synthesis generates a syntax error for detection of mixed blocking and non-blocking assignments within an `ALWAYS` blocks.

Avoid Unconnected, Non-Existent Ports

Other Quartus software products allow you to instantiate and name an unconnected port that does not exist in the module. Quartus Prime Pro Edition synthesis requires that a ports exists in the module prior to instantiation and naming. Modify your RTL to match this requirement.

To avoid syntax errors, remove all unconnected and non-existent ports for Quartus Prime Pro Edition migration.

Table 1-10: Unconnected, Non-Existent Port Differences

Other Quartus Software Products[3]	Quartus Prime Pro Edition
Synthesis allows you to instantiate and name unconnected or non-existent ports that do not exist on the module.	Synthesis generates a syntax error for detection of mixed blocking and non-blocking assignments within an `ALWAYS` blocks.

Avoid Illegal Parameter Ranges

Other Quartus software products allow constant numeric (integer or floating point) values for parameters that exceed the language specifications. Quartus Prime Pro Edition synthesis generates an error for detection of constant numeric (integer or floating point) parameter values that exceed the language specification. To avoid syntax errors, ensure that constant numeric (integer or floating point) values for parameters conform to the language specifications.

Update Verilog and VHDL Type Mapping

Other Quartus software products map "true and "false" strings in Verilog HDL to TRUE and FALSE Boolean values in VHDL. Quartus Prime Pro Edition synthesis requires that you use `0` for "`false`" and `1` for "`true`" in Verilog HDL files (**.v**). Quartus Prime Pro Edition synthesis generates an error for detection of non-Verilog constructs in **.v** files. To avoid syntax errors, ensure that your RTL accommodates these standards.

[3] For brevity, this section refers to Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software collectively as "other Quartus software products."

Document Revision History

Table 1-11: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• First version of document.

Managing Quartus Prime Projects 2

2015.11.02

The Quartus Prime 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 create a new project quickly and specify basic project settings.

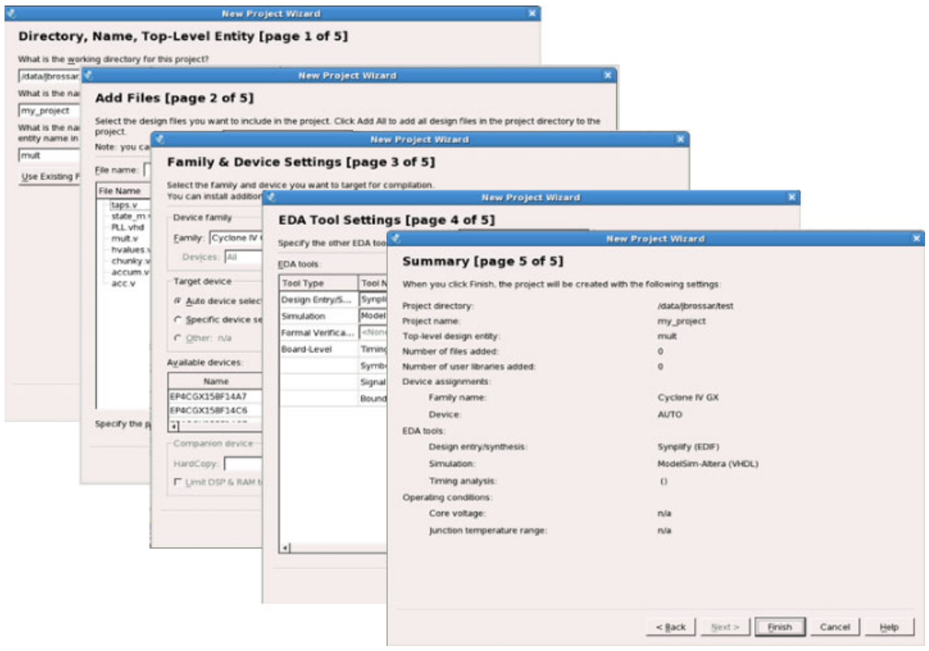
When you open a project, a unified GUI displays integrated project information. The Project Navigator allows you to view and edit the elements of your project. The Messages window lists important information about project processing.

You can save multiple revisions of your project to experiment with settings that achieve your design goals. Quartus Prime 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]
Summary [page 5 of 5]
< Back
Next >
Finish
Cancel
Help

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 Prime Projects

A single Quartus Prime Project File (**.qpf**) represents each project. The text-based **.qpf** references the Quartus Prime 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 2-1: Quartus Prime Project Files

File Type	Contains	To Edit	Format
Project file	Project and revision name	**File** > **New Project Wizard**	Quartus Prime Project File (**.qpf**)
Project settings	Lists design files, entity settings, target device, synthesis directives, placement constraints	**Assignments** > **Settings**	Quartus Prime Settings File (**.qsf**)
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
Programming 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 Prime IP File (**.qip**)
Qsys system files	Qsys system and IP core files	**Tools** > **Qsys**	Qsys System File (**.qsys**)
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**)

File Type	Contains	To Edit	Format
Archive files	Complete project as single compressed file	**Project** > **Archive Project**	Quartus Prime Archive File (**.qar**)

Project Management Best Practices

The Quartus Prime software provides various options for setting up a project. The following best practices help ensure efficient management and portability of your project files.

Setting and Project File Best Practices

- Avoid manually editing Quartus Prime data files, such as the Quartus Prime Project File (**.qpf**), Quartus Prime Settings File (**.qsf**), Quartus IP File (**.qip**), or Qsys System File (**.qsys**). Typos in these files can cause software errors. For example, the software may ignore settings and assignments.

  Every Quartus Prime project revision automatically includes a supporting **.qpf** that preserves various project settings and constraints that you enter in the GUI or add with Tcl commands. This file contains basic information about the current software version, date, and project-wide and entity level settings. Due to dependencies between the **.qpf** and **.qsf**, avoid manually editing **.qsf** files.
- Do not compile multiple projects into the same directory. Instead, use a separate directory for each project.
- By default, the Quartus Prime software saves all project output files, such as Text-Format Report Files (**.rpt**), in the project directory. Instead of manually moving project output files, change your project compilation settings to save them in a separate directory.

  To save these files into a different directory choose **Assignments** > **Settings**. Turn on the **Save project output files in specified directory** option and specify a directory for the output files.

Project Archive and Source Control Best Practices

- Click **Project** > **Archive Project** to archive your project for revision control.

  As you develop your design, your Quartus Prime project directory contains a variety of source and settings files, compilation database files, output, and report files. You can archive these files using the Archive feature and save the archive for later use or place it under revision control.

  1. Choose **Project** > **Archive Project** > **Advanced** to open the **Advanced Archive Settings** dialog box.
  2. Choose a file set to archive. For example, choose **File set** > **Source control with incremental compilation and Rapid Recompile database** to save the source and database file required to re-create your project with your Rapid Recompile revisions.
  3. Add additional files by clicking **Add** (optional).

  To restore your archived project, choose **Project** > **Restore Archived Project**. Restore your project into a new, empty directory.

IP Core Best Practices

- Do not manually edit or write your own **.qsys** or **.qip** file. Use the Quartus Prime software tools to create and edit these files.

  Note: When generating IP cores, do not generate files into a directory that has a space in the directory name or path.
- When you generate an IP core using the IP Catalog, the Quartus Prime software generates a **.qsys** (for Qsys-generated IP cores) or **.qip** file. Always add the generated **.qsys** or **.qip** to your project. Do not add the parameter editor generated file (**.v** or **.vhd**) to your design without the **.qsys** or **.qip** file. Otherwise, you cannot use the IP upgrade or IP parameter editor feature.

  Note: For Qsys-generated IP cores, adding the **.qsys** file to the project instead of the **.qip** file simplifies modifying the IP with the parameter editor.
- Plan your directory structure ahead of time. Do not change the relative path between a **.qsys** file and it's generation output directory. If you must move the **.qsys** file, ensure that the generation output directory remains with the **.qsys** file.
- Do not add IP core files directly from the **/quartus/libraries/megafunctions** directory in your project. Otherwise, you must update the files for each subsequent software release. Instead, use the IP Catalog and then add the **.qip** to your project.
- Do not use IP files that the Quartus Prime software generates for RAM or FIFO blocks targeting older device families (even though the Quartus Prime software does not issue an error).
- When generating a ROM function, save the resulting **.mif** or **.hex** file in the same folder as the corresponding IP core's **.qsys** or **.qip** file. For example, moving all of your project's **.mif** or **.hex** files to the same directory causes relative path problems after archiving the design.
- Always use the Quartus Prime `ip-setup-simulation` and `ip-make-simscript` utilities to generate simulation scripts for each IP core or Qsys system in your design. These utilities produce a single simulation script that does not require manual update for upgrades to Quartus Prime software or IP versions. Refer to *Generating Version-Independent IP and Qsys Simulation Scripts* for details.

Related Information

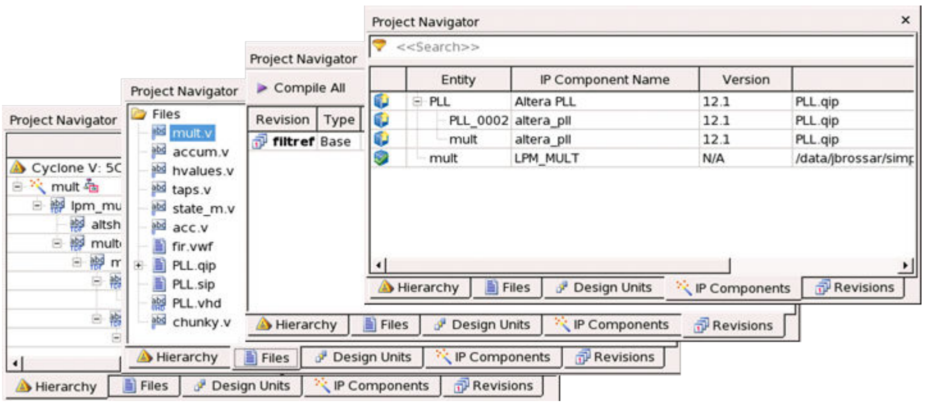
Generating Version-Independent IP and Qsys Simulation Scripts on page 2-35

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 2-1: Project Navigator Hierarchy, Files, Revisions, and IP



Revision	Type
filtref	Base

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

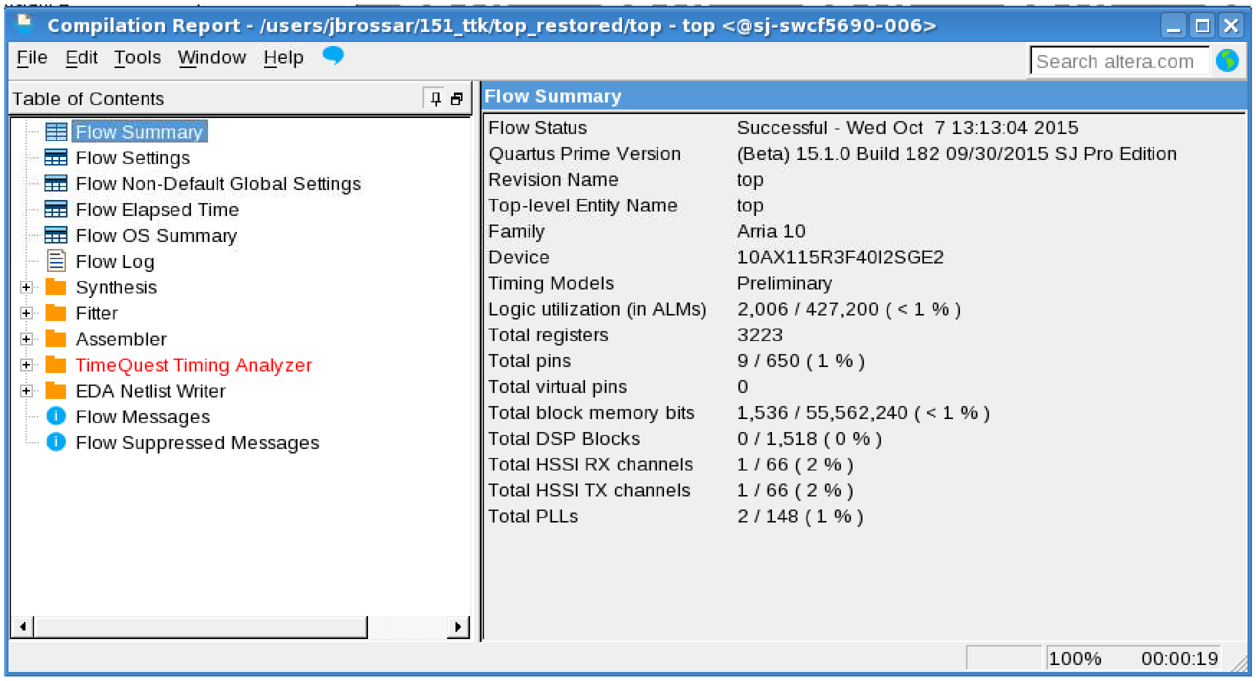
Viewing Project Reports

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

- 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 2-2: Report Panel



Flow Summary

Flow Status	Successful - Wed Oct 7 13:13:04 2015
Quartus Prime Version	(Beta) 15.1.0 Build 182 09/30/2015 SJ Pro Edition
Revision Name	top
Top-level Entity Name	top
Family	Arria 10
Device	10AX115R3F40I2SGE2
Timing Models	Preliminary
Logic utilization (in ALMs)	2,006 / 427,200 (< 1 %)
Total registers	3223
Total pins	9 / 650 (1 %)
Total virtual pins	0
Total block memory bits	1,536 / 55,562,240 (< 1 %)
Total DSP Blocks	0 / 1,518 (0 %)
Total HSSI RX channels	1 / 66 (2 %)
Total HSSI TX channels	1 / 66 (2 %)
Total PLLs	2 / 148 (1 %)

Related Information

List of Compilation Reports

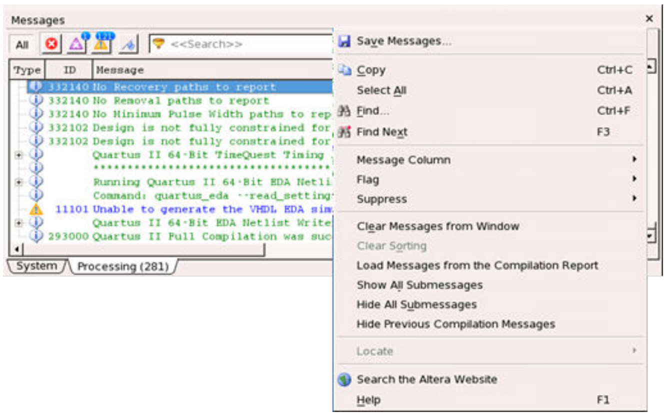
Viewing Project Messages

The Messages window (**View** > **Utility Windows** > **Messages**) displays information, warning, and error messages about Quartus Prime 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.

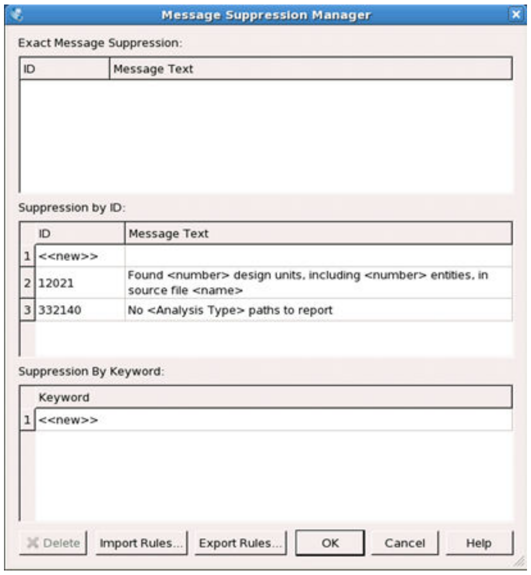
Figure 2-3: Messages Window



```
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 2-4: Message Suppression by Message ID Number



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.

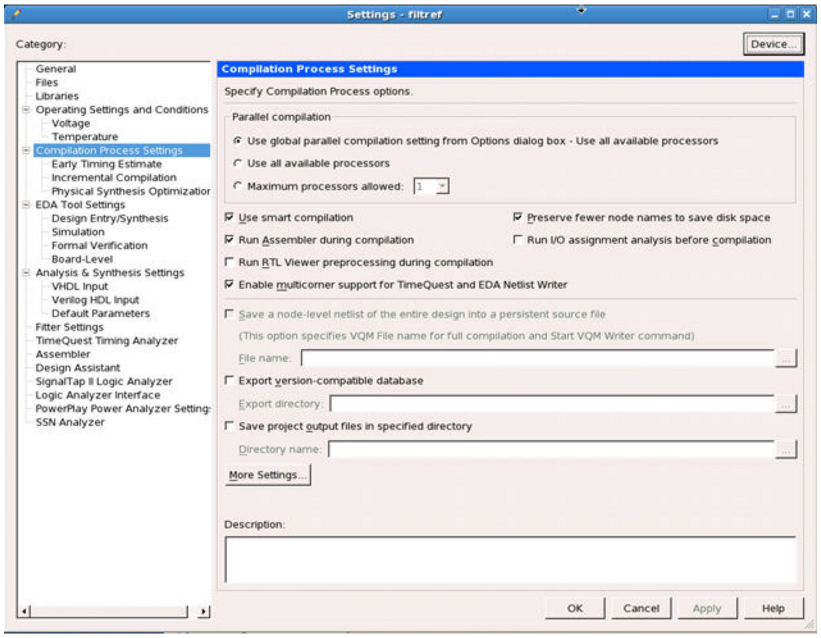
The **.qsf** stores each revision's project settings.

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

Figure 2-5: Settings Dialog Box for Global Project Settings



Compilation Process Settings

Specify Compilation Process options.

Parallel compilation

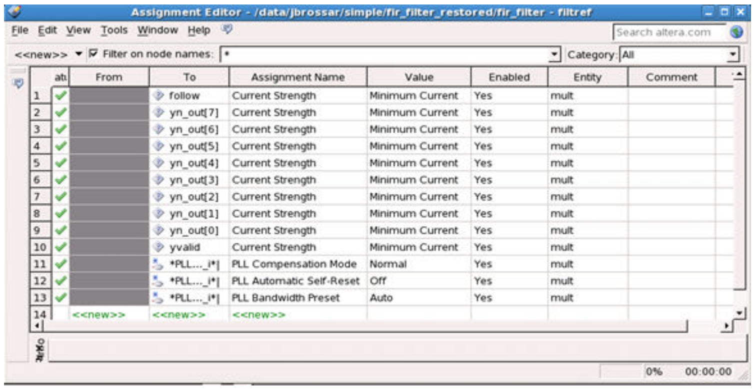
[● Use global parallel compilation setting from Options dialog box - Use all available processors]
○ Use all available processors
○ Maximum processors allowed: 1

[✓] Use smart compilation
[✓] Preserve fewer node names to save disk space
[✓] Run Assembler during compilation
[ ] Run I/O assignment analysis before compilation
[ ] Run RTL Viewer preprocessing during compilation
[✓] Enable multicorner support for TimeQuest and EDA Netlist Writer
[ ] 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:

Description:

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

Figure 2-6: Assignment Editor Spreadsheet


Assignment Editor - /data/jbrossar/simple/fir_filter_restored/fir_filter - filtref
File Edit View Tools Window Help
Search altera.com
<<new>>
Filter on node names: *
Category: All
0%
00:00:00

	ati	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>>				

Optimizing Project Settings

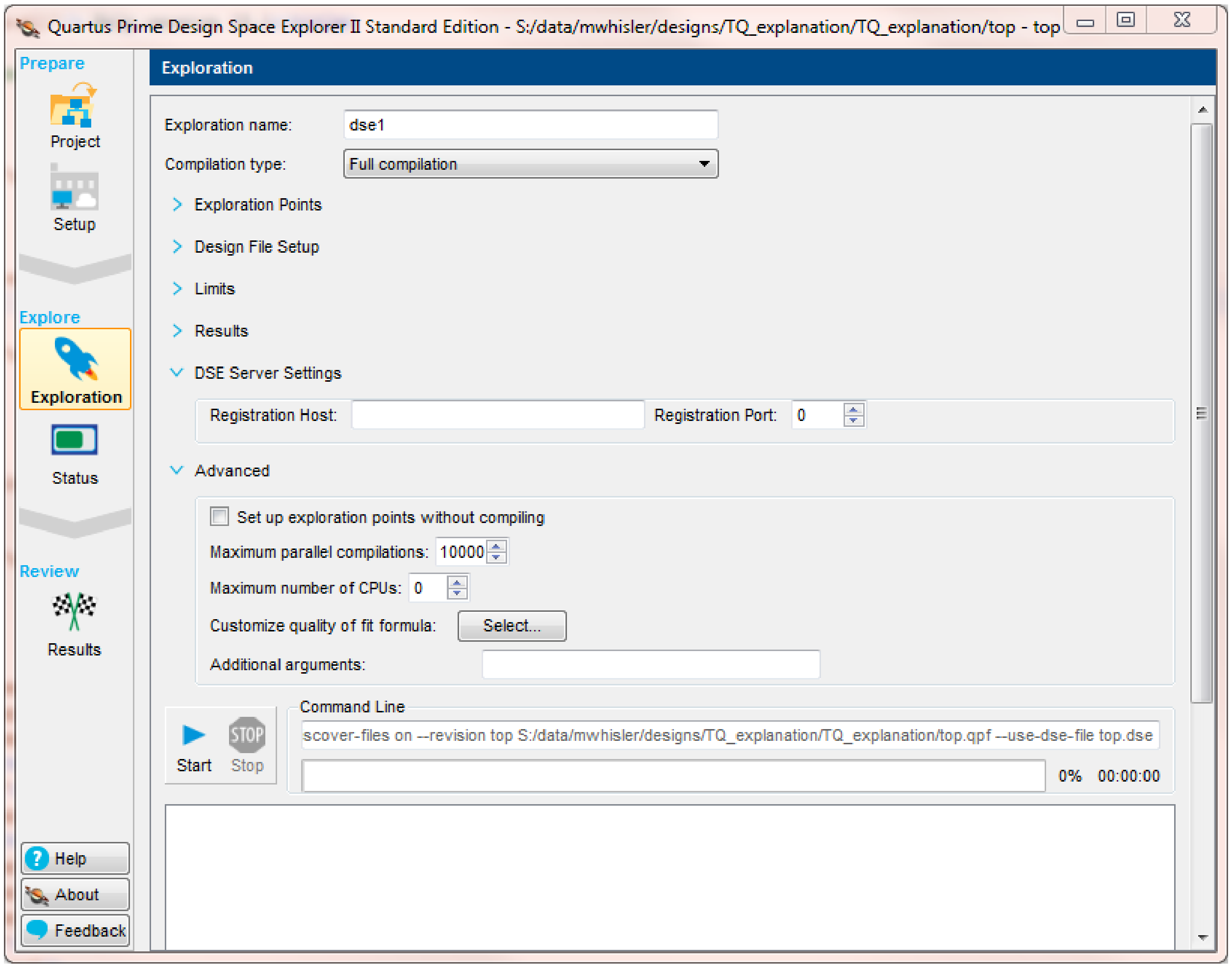
Optimize project settings to meet your design goals. The Quartus Prime Design Space Explorer II iteratively compiles your project with various setting combinations to find the optimal setting for your goals. Alternatively, you can create a project revision or project copy to manually compare various 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 2-7: Design Space Explorer II



Exploration name: dse1
Compilation type: Full compilation
Registration Host:
Registration Port: 0
Set up exploration points without compiling
Maximum parallel compilations: 10000
Maximum number of CPUs: 0
Customize quality of fit formula:
Additional arguments:
Command Line
scover-files on --revision top S:/data/mwhisler/designs/TQ_explanation/TQ_explanation/top.qpf --use-dse-file top.dse
0% 00:00:00

Optimizing with Project Revisions

You can save multiple, named project revisions within your Quartus Prime 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:

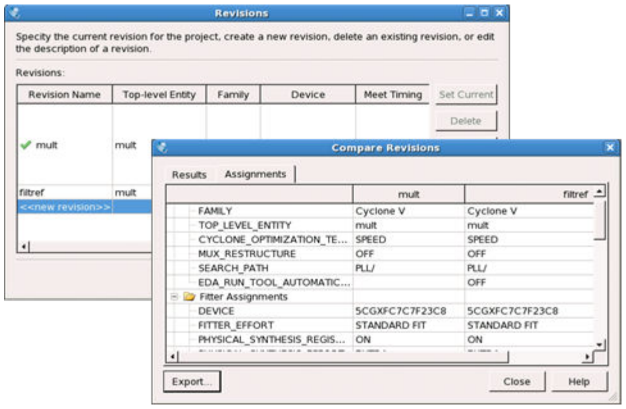
- Create a unique revision to optimize a design for different criteria, such as by area in one revision and by f_{MAX} in another revision.
- When you create a new revision the default Quartus Prime settings initially apply.
- Create a revision of a revision to experiment with settings and constraints. 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 Prime 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 2-8: 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.

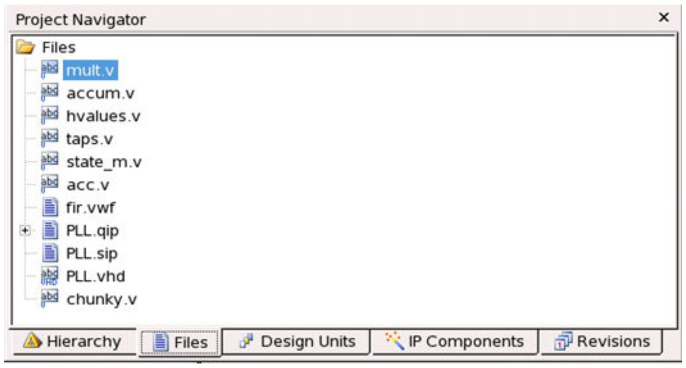
Managing Logic Design Files

The Quartus Prime 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 Prime software includes full-featured schematic and text editors, as well as HDL templates to accelerate your design work. The Quartus Prime software supports VHDL Design Files (**.vhd**), Verilog HDL Design Files (**.v**), SystemVerilog (**. sv**) and schematic Block Design Files (**. bdf**). 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 2-9: 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 Prime projects. The**.qsf** stores project library information.

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

Related Information

Design Library Migration Guidelines on page 2-48

Specifying Design Libraries

To specify project libraries from the GUI:

1. Click **Assignment** > **Settings**.
2. Click **Libraries** and specify 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 10-1
- Recommended HDL Coding Styles on page 11-1

Managing Timing Constraints

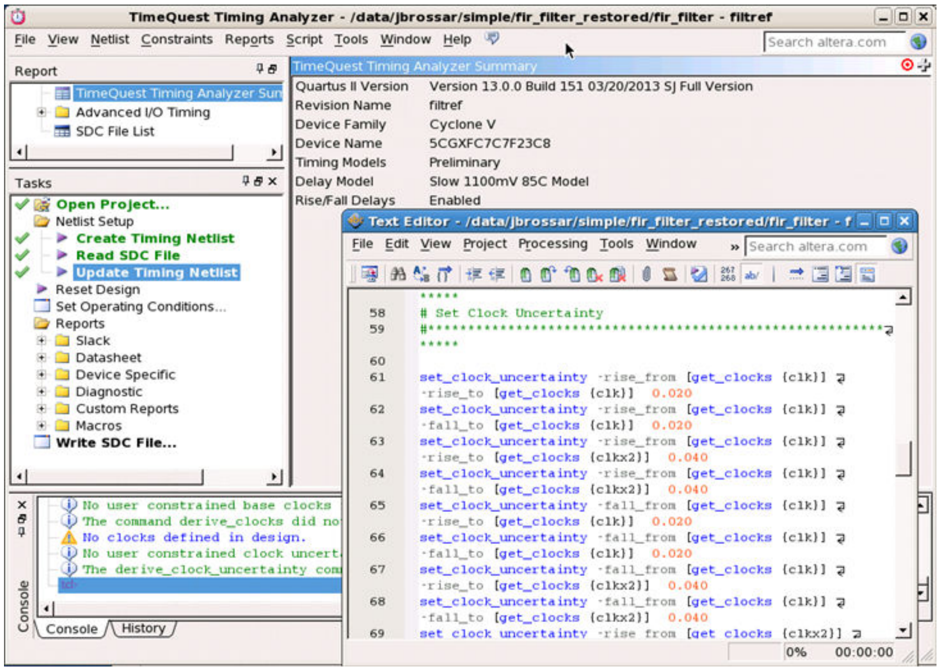
View basic information about your project in the Project Navigator, Report panel, and Messages window.

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 2-10: 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...
TimeQuest Timing Analyzer Summary
Text Editor - /data/jbrossar/simple/fir_filter_restored/fir_filter - f
File Edit View Project Processing Tools Window
Search altera.com
Console
History
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

```
*****
# 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}] 
```

```
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>
```

Related Information

Quartus Prime TimeQuest Timing Analyzer

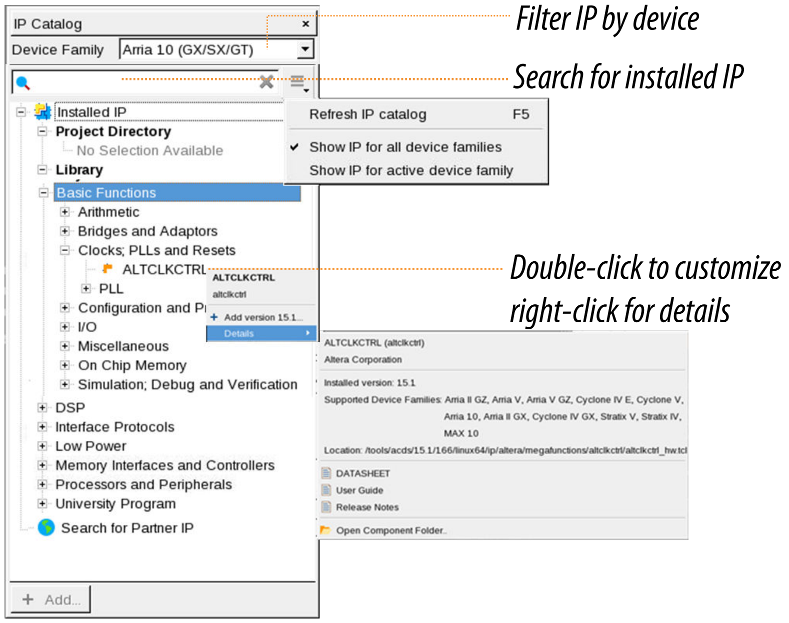
Introduction to Altera IP Cores

Altera® and strategic IP partners offer a broad portfolio of configurable IP cores optimized for Altera devices. The Quartus Prime 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. The Quartus Prime 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 types of IP cores:

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

IP Catalog and Parameter Editor

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



ALTCLKCTRL (altclkctrl)
Altera Corporation
Installed version: 15.1
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/15.1/166/linux64/ip/altera/megafunctions/altclkctrl/altclkctrl_hw.tcl

The IP Catalog displays the installed IP cores available for your design. Double-click any IP core to launch the parameter editor and generate files representing your IP variation. 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 click links to IP documentation.
- Click **Search for Partner IP**, to access partner IP information on the Altera website.

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 Prime IP file (**.qip**) representing the IP core in your project. You can also parameterize an IP variation without an open project.

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 Prime IP Catalog. For more information about using the Qsys IP Catalog, refer to *Creating a System with Qsys* in the *Quartus Prime Handbook*.

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.

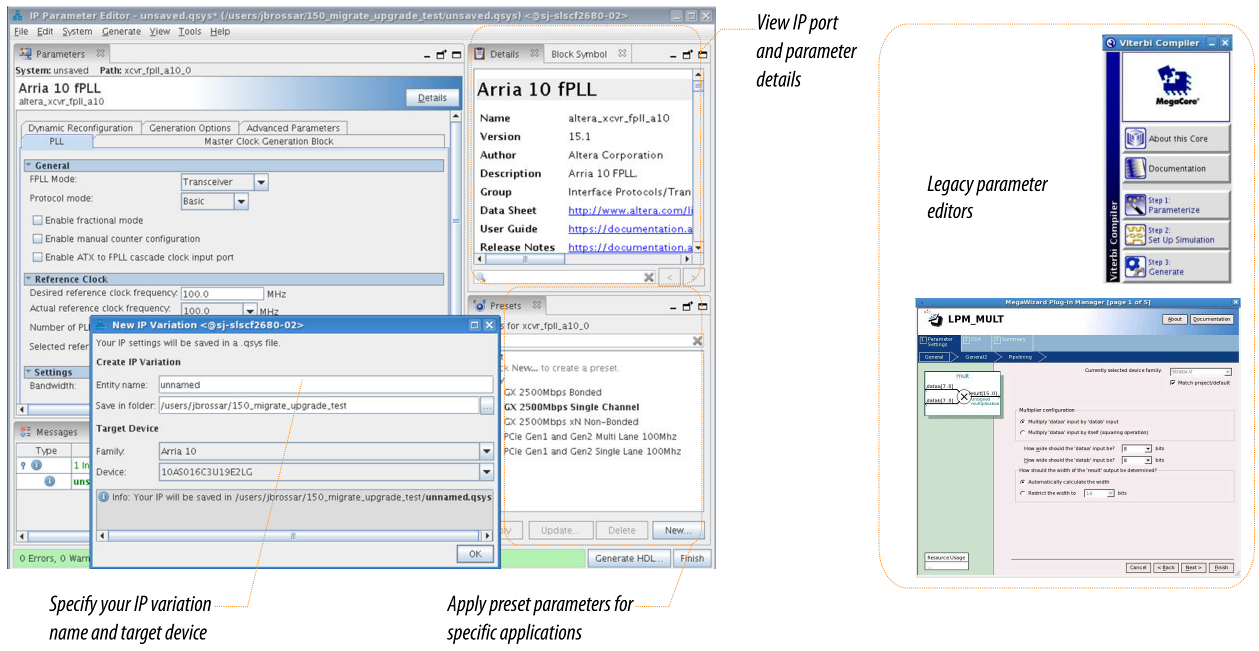
Related Information

Creating a System With Qsys on page 4-1

Using the Parameter Editor

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

Figure 2-11: 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
Arria 10 fPLL
Viterbi Compiler
LPM_MULT

Name altera_xcvr_fpll_a10
Version 15.1
Author Altera Corporation
Description Arria 10 FPLL
Group Interface Protocols/Tran
Data Sheet http://www.altera.com/li
User Guide https://documentation.a
Release Notes https://documentation.a

New IP Variation <@sj-slscf2680-02>
Your IP settings will be saved in a .qsys file.
Create IP Variation
Entity name: unnamed
Save in folder: /users/jbrossar/150_migrate_upgrade_test
Target Device
Family: Arria 10
Device: 10AS016C3U19E2LG
Info: Your IP will be saved in /users/jbrossar/150_migrate_upgrade_test/unnamed.qsys

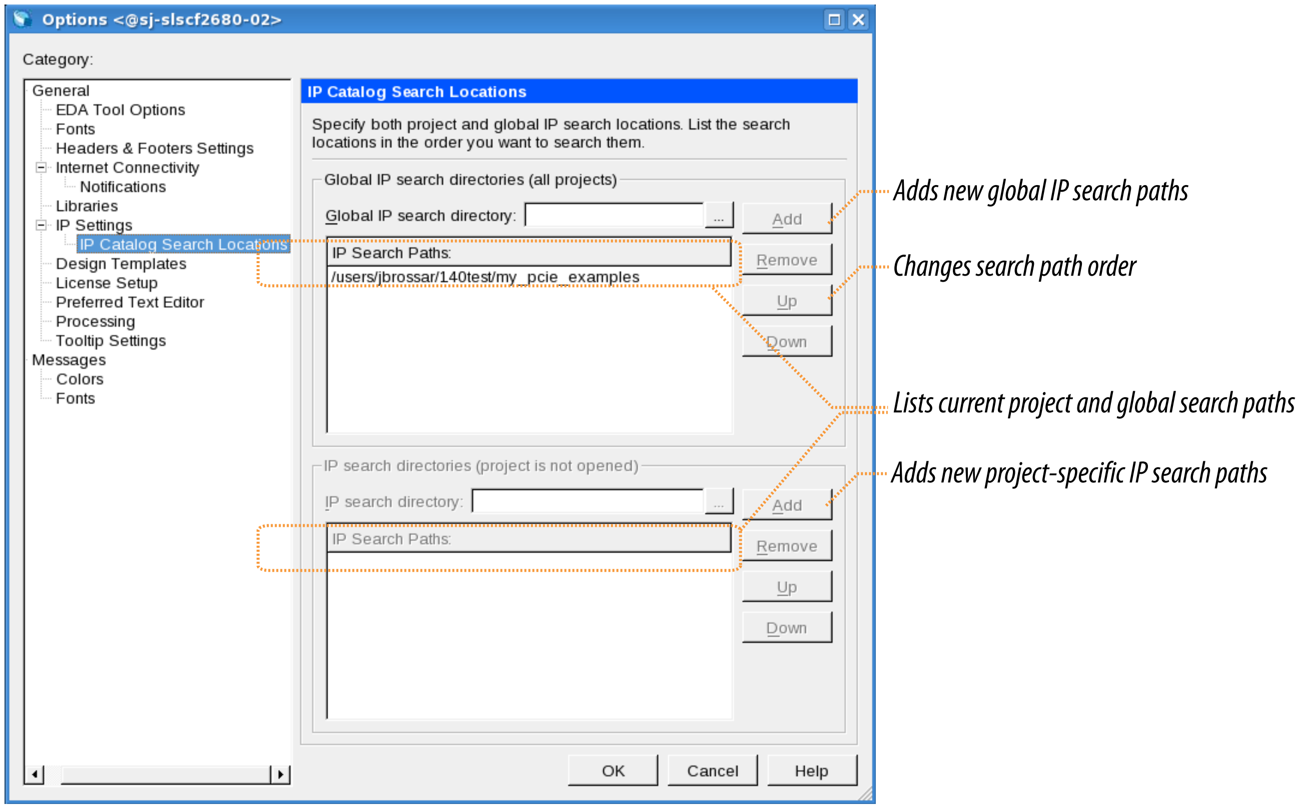
- Use preset settings in the parameter editor (where provided) to instantly apply preset parameter values for specific applications.
- View port and parameter descriptions, and click links to documentation.
- Generate testbench systems or example designs (where provided).

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.

Figure 2-12: Specifying IP Search Locations


Options <@sj-slscf2680-02>
Category:
General
EDA Tool Options
Fonts
Headers & Footers Settings
Internet Connectivity
Notifications
Libraries
IP Settings
IP Catalog Search Locations
Design Templates
License Setup
Preferred Text Editor
Processing
Tooltip Settings
Messages
Colors
Fonts
IP Catalog Search Locations
Global IP search directories (all projects)
Global IP search directory:
Add
Remove
Up
Down
IP Search Paths:
IP search directories (project is not opened)
IP search directory:
Add
Remove
Up
Down
IP Search Paths:
OK
Cancel
Help
Adds new global IP search paths
Changes search path order
Lists current project and global search paths
Adds new project-specific IP search paths

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

/users/jbrossar/140test/my_pcie_examples

The Quartus Prime 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 Prime 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 2-2: IP Search Locations

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

If the Quartus Prime 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 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 update the IP Catalog by clicking **Refresh IP Catalog** in the drop-down list. In Qsys, click **File** > **Refresh System** to update the IP Catalog.

General Settings for IP

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

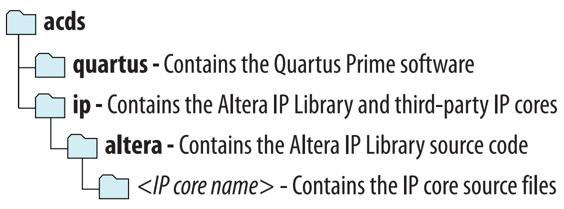
Table 2-3: IP Core General Setting Locations

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 Prime 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 regenerate 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 Prime software searches for IP cores in the project directory, in the Altera installation directory, and in the IP search path.

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 Prime software. After you are satisfied with functionality and performance, visit the Self Service Licensing Center to obtain a license number for any Altera product.

Figure 2-13: IP Core Installation Path



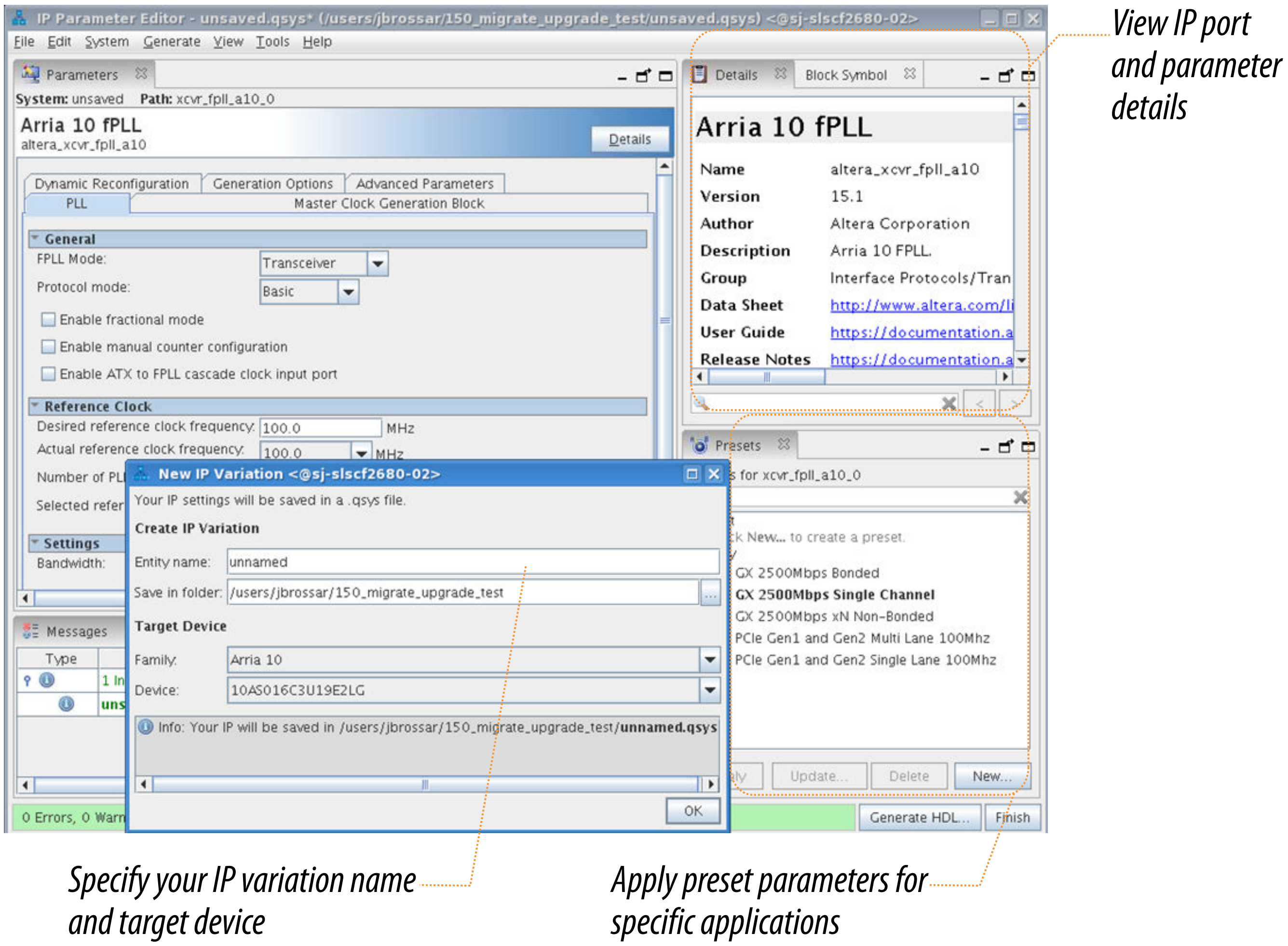
- **acds**
  - **quartus -** Contains the Quartus Prime software
  - **ip -** Contains the Altera IP Library and third-party IP cores
    - **altera -** Contains the Altera IP Library source code
      - *<IP core name>* - Contains the IP core source files

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

Generating IP Cores

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.

Figure 2-14: IP Parameter Editor


IP Parameter Editor - unsaved.qsys* (/users/jbrossar/150_migrate_upgrade_test/unsaved.qsys) <@sj-slscf2680-02>
File Edit System Generate View Tools Help
Parameters
Details
Block Symbol
Presets
Messages
View IP port and parameter details
Specify your IP variation name and target device
Apply preset parameters for specific applications
Update...
Delete
New...
Generate HDL...
Finish
OK

System: unsaved Path: xcvr_fpll_a10_0

Arria 10 fPLL
altera_xcvr_fpll_a10

Dynamic Reconfiguration | Generation Options | Advanced Parameters
PLL | Master Clock Generation Block

General
FPLL Mode: Transceiver
Protocol mode: Basic
Enable fractional mode
Enable manual counter configuration
Enable ATX to FPLL cascade clock input port

Reference Clock
Desired reference clock frequency: 100.0 MHz
Actual reference clock frequency: 100.0 MHz
Number of PLL
Selected refer

Settings
Bandwidth:

Arria 10 fPLL

Name	altera_xcvr_fpll_a10
Version	15.1
Author	Altera Corporation
Description	Arria 10 FPLL
Group	Interface Protocols/Tran
Data Sheet	http://www.altera.com/li
User Guide	https://documentation.a
Release Notes	https://documentation.a

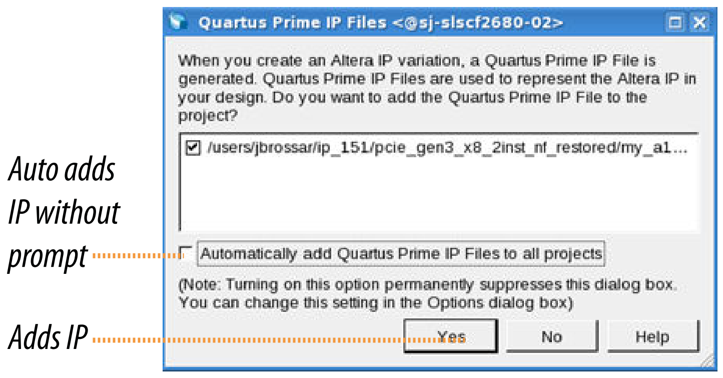
New IP Variation <@sj-slscf2680-02>
Your IP settings will be saved in a .qsys file.
Create IP Variation
Entity name: unnamed
Save in folder: /users/jbrossar/150_migrate_upgrade_test
Target Device
Family: Arria 10
Device: 10AS016C3U19E2LG
Info: Your IP will be saved in /users/jbrossar/150_migrate_upgrade_test/**unnamed.qsys**

s for xcvr_fpll_a10_0
k New... to create a preset.
GX 2500Mbps Bonded
GX 2500Mbps Single Channel
GX 2500Mbps xN Non-Bonded
PCIe Gen1 and Gen2 Multi Lane 100Mhz
PCIe Gen1 and Gen2 Single Lane 100Mhz

Type
1 In
uns
0 Errors, 0 Warn

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**. Do not include spaces in IP variation names or paths.
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**. Click **Yes** if prompted to add files representing the IP variation to your project. Optionally turn on the option to **Automatically add Quartus Prime IP Files to All Projects**. Click **Project** > **Add/Remove Files in Project** to add IP files at any time.

Figure 2-15: Adding IP Files to Project


Quartus Prime IP Files <@sj-slscf2680-02>
Auto adds IP without prompt
Adds IP
Yes
No
Help

When you create an Altera IP variation, a Quartus Prime IP File is generated. Quartus Prime IP Files are used to represent the Altera IP in your design. Do you want to add the Quartus Prime IP File to the project?

☑ /users/jbrossar/ip_151/pcie_gen3_x8_2inst_nf_restored/my_a1...

☐ Automatically add Quartus Prime IP Files to all projects

(Note: Turning on this option permanently suppresses this dialog box. You can change this setting in the Options dialog box)

The generated **.qsys** file must be added to your project to represent IP and Qsys systems.

9. After generating and instantiating your IP variation, make appropriate pin assignments to connect ports.

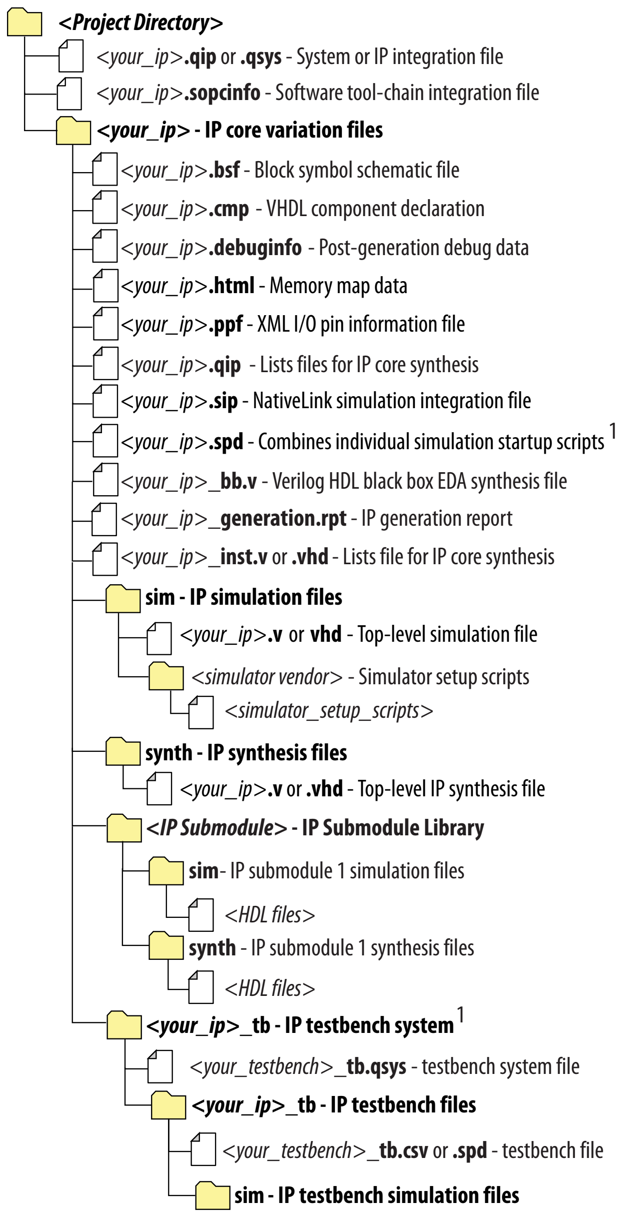
Related Information

- IP User Guide Documentation
- Altera IP Release Notes

Files Generated for Altera IP Cores and Qsys Systems

The Quartus Prime software generates the following output file structure for IP cores and Qsys systems. The generated **.qsys** file must be added to your project to represent IP and Qsys systems.

Figure 2-16: Files generated for IP cores and Qsys Systems


<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>.bsf - Block symbol schematic file
<your_ip>.cmp - VHDL component declaration
<your_ip>.debuginfo - Post-generation debug data
<your_ip>.html - Memory map data
<your_ip>.ppf - XML I/O pin information file
<your_ip>.qip - Lists files for IP core synthesis
<your_ip>.sip - NativeLink simulation integration file
<your_ip>.spd - Combines individual simulation startup scripts 1
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_generation.rpt - IP generation report
<your_ip>_inst.v or .vhd - Lists file for IP core synthesis
sim - IP simulation files
<your_ip>.v or vhd - Top-level simulation file
<simulator vendor> - Simulator setup scripts
<simulator_setup_scripts>
synth - IP synthesis files
<your_ip>.v or .vhd - Top-level IP synthesis file
<IP Submodule> - IP Submodule Library
sim- IP submodule 1 simulation files
<HDL files>
synth - IP submodule 1 synthesis files
<HDL files>
<your_ip>_tb - IP testbench system 1
<your_testbench>_tb.qsys - testbench system file
<your_ip>_tb - IP testbench files
<your_testbench>_tb.csv or .spd - testbench file
sim - IP testbench simulation files

1. If supported and enabled for your IP core variation.

Table 2-4: IP Core and Qsys Simulation Generated Files

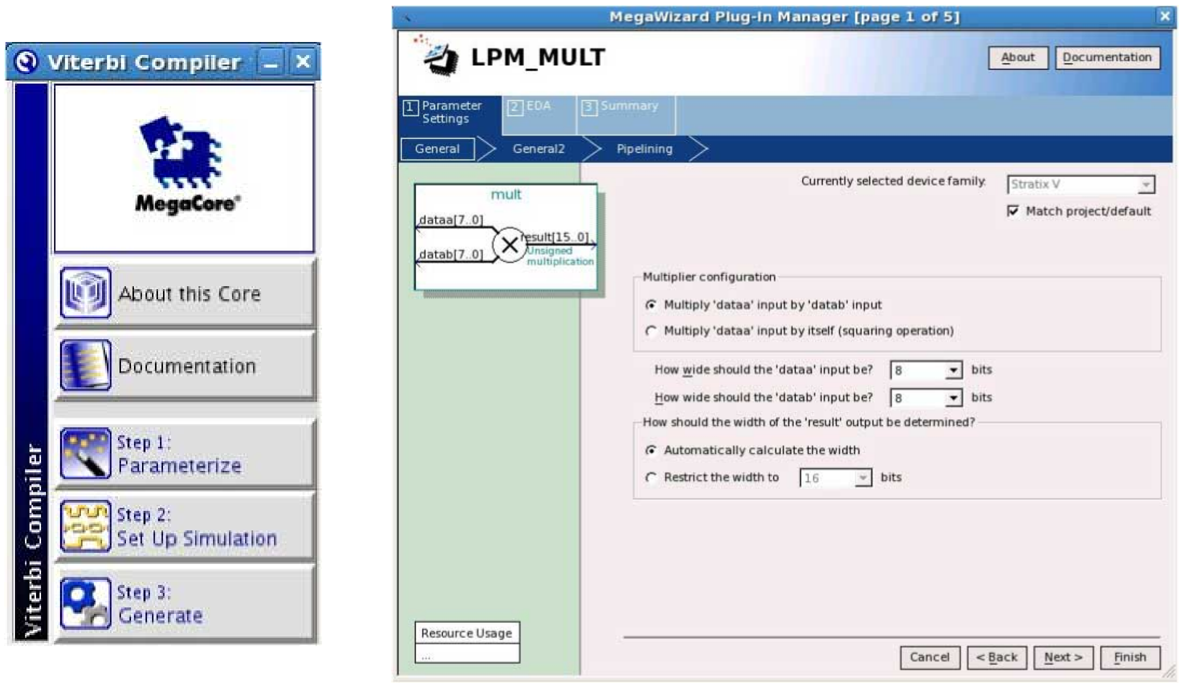
File Name	Description
<*my_ip*>.qsys	The Qsys system or top-level IP variation file. <*my_ip*> is the name that you give your IP variation.
<*system*>.sopcinfo	Describes the connections and IP component parameterizations in your Qsys system. You can parse its contents to get requirements when you develop 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.
<*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 Prime 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 Prime 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*>_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.

File Name	Description
<*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.
<*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.
<*IP submodule*>/	For each generated IP submodule directory, Qsys generates **/synth** and **/sim** sub-directories.

Generating IP Cores (Legacy 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.

Figure 2-17: 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

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.

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**. Do not include spaces in IP variation names or paths.
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: For devices released prior to Arria 10 devices, the generated **.qip** and **.sip** files must be added to your project to represent IP and Qsys systems. 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.

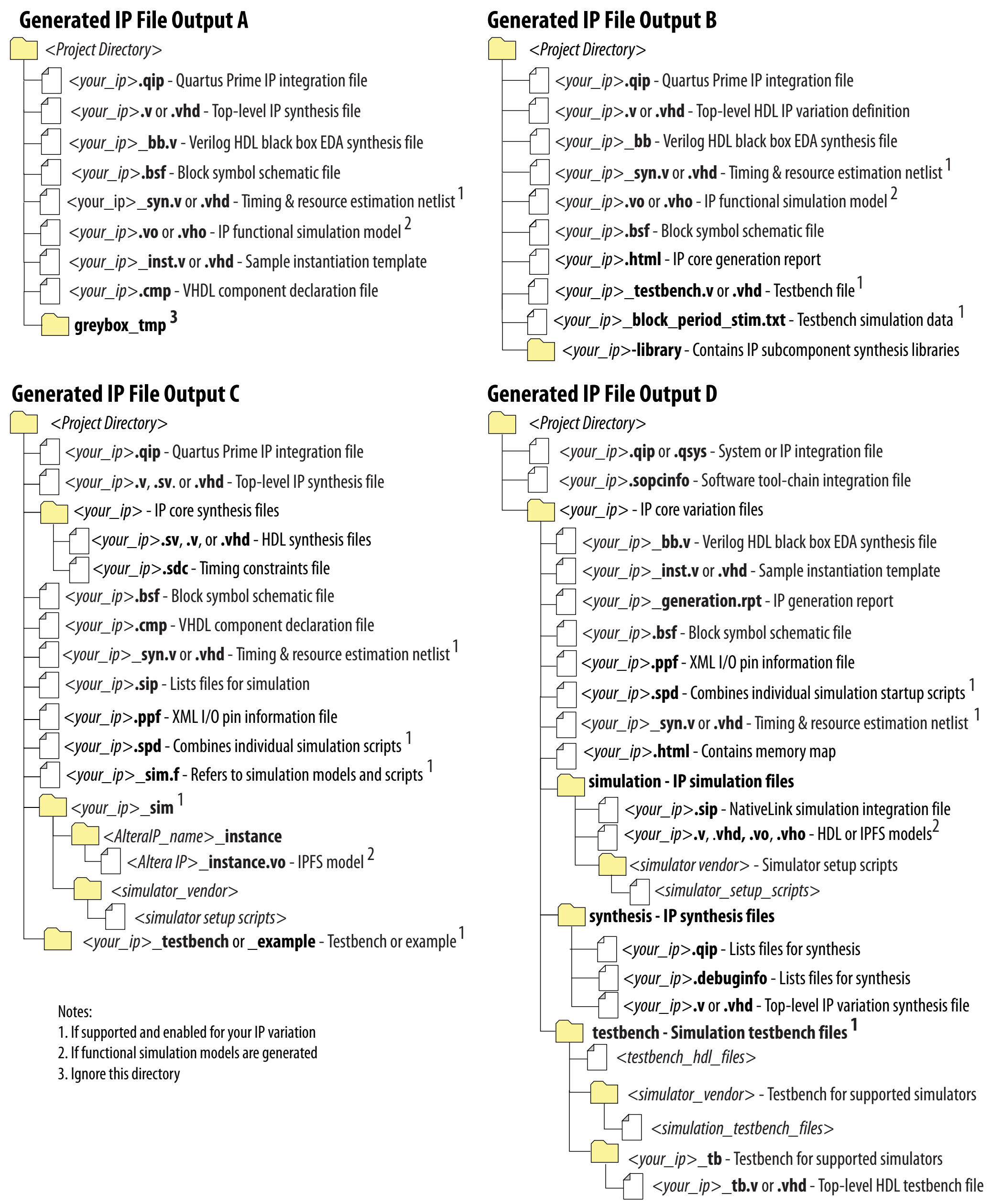
Related Information

- IP User Guide Documentation
- Altera IP Release Notes

Files Generated for Altera IP Cores (Legacy Parameter Editors)

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

Figure 2-18: IP Core Generated Files (Legacy Parameter Editors)


Generated IP File Output A
<Project Directory>
<your_ip>.qip - Quartus Prime 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 Prime 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 Prime 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
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
Notes:
1. If supported and enabled for your IP variation
2. If functional simulation models are generated
3. Ignore this directory

Note: For devices released prior to Arria 10 devices, the generated **.qip** and **.sip** files must be added to your project to represent IP and Qsys systems. To manually add an IP variation to a Quartus Prime 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.

Modifying an IP Variation

After generating an IP core variation, you can modify it's parameters in the parameter editor. Use any of the following methods to modify an IP variation in the parameter editor.

Table 2-5: 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.
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.

Scripting IP Core Generation

You can use the `qsys-script` and `qsys-generate` utilities to define and generate an IP core variation outside of the Quartus Prime GUI.

To parameterize and generate an IP core at the command-line, follow these steps:

1. Run `qsys-script` to execute a Tcl script that instantiates the IP and sets desired parameters:

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

2. Run `qsys-generate` to generate the IP core 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 2-6: 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 Prime 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.

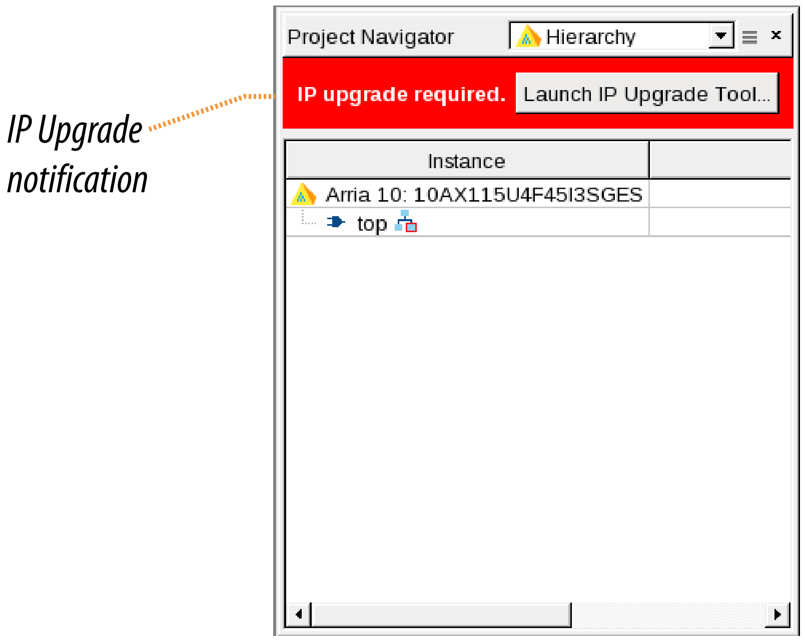
Option	Usage	Description
`--simulation=<VERILOG|VHDL>`	Optional	Creates a simulation model for the Qsys system. The simulation model contains generated HDL files for the simulator, and may include simulation-only features. You must specify the preferred simulation language.
`--testbench=<SIMPLE|STANDARD>`	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.
`--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.

For command-line help listing all options for these executables, type <*executable name*> `--help`

Upgrading IP Cores

IP core variants generated with a previous version or different edition of the Quartus Prime software may require upgrading before use in the current version or edition of the Quartus Prime software. When you open a project containing outdated IP, the Project Navigator displays a banner indicating the IP upgrade status. Click **Launch IP Upgrade Tool**, or **Project** > **Upgrade IP Components** to upgrade outdated IP cores.

Figure 2-19: IP Upgrade Alert in Project Navigator



Icons in the **Upgrade IP Components** dialog box indicate when IP upgrade is required, optional, or unsupported for IP cores in your design. You must upgrade IP cores that require upgrade before you can compile the IP variation in the current version of the Quartus Prime software.

The upgrade process preserves the original IP variation file in the project directory as <*my_variant*>_**BAK.qsys**.

Note: Upgrading IP cores 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 Prime file; such as the Quartus Prime Settings File (**.qsf**), Synopsys Design Constraints File (**.sdc**), or SignalTap File (**.stp**), if these files contain instance names. The Quartus Prime software reads only the instance name and ignores the entity name in paths that specify both names. Use only instance names in assignments.

Table 2-7: 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 Prime 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 Required	You must upgrade the IP variation before compiling in the current version of the Quartus Prime 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 Prime software due to incompatibility with the current version of the Quartus Prime 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 Prime software.
IP Upgrade Mismatch Warning	Warning of non-critical IP core differences in migrating IP to another device family.

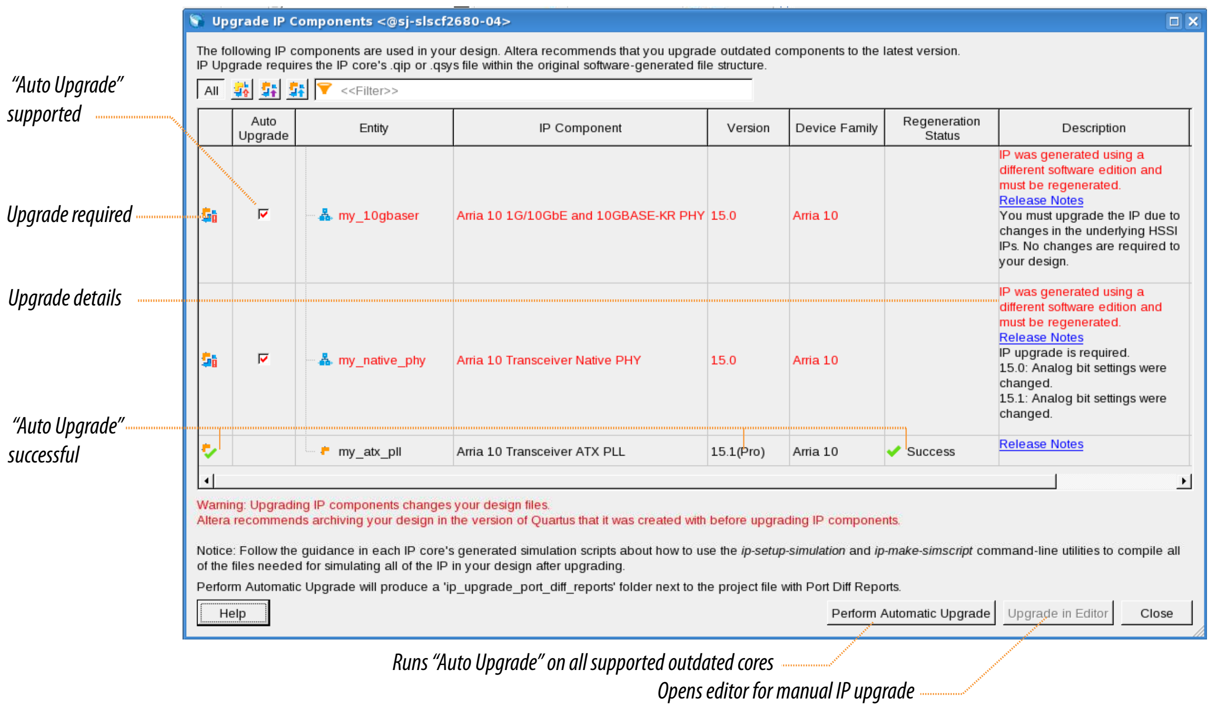
Follow these steps to upgrade IP cores:

1. In the latest version of the Quartus Prime software, open the Quartus Prime 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 2-20: Upgrading IP Cores


"Auto Upgrade" supported
Upgrade required
Upgrade details
"Auto Upgrade" successful
Runs "Auto Upgrade" on all supported outdated cores
Opens editor for manual IP upgrade
Upgrade IP Components <@sj-slscf2680-04>
All
<<Filter>>
Help
Perform Automatic Upgrade
Upgrade in Editor
Close

The following IP components are used in your design. Altera recommends that you upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original software-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description
	✓	my_10gbaser	Arria 10 1G/10GbE and 10GBASE-KR PHY	15.0	Arria 10		IP was generated using a different software edition and must be regenerated. Release Notes You must upgrade the IP due to changes in the underlying HSSI IPs. No changes are required to your design.
	✓	my_native_phy	Arria 10 Transceiver Native PHY	15.0	Arria 10		IP was generated using a different software edition and must be regenerated. Release Notes IP upgrade is required. 15.0: Analog bit settings were changed. 15.1: Analog bit settings were changed.
✓		my_atx_pll	Arria 10 Transceiver ATX PLL	15.1(Pro)	Arria 10	✓ Success	Release Notes

Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design in the version of Quartus that it was created with before upgrading IP components.

Notice: Follow the guidance in each IP core's generated simulation scripts about how to use the *ip-setup-simulation* and *ip-make-simscript* command-line utilities to compile all of the files needed for simulating all of the IP in your design after upgrading.

Perform Automatic Upgrade will produce a 'ip_upgrade_port_diff_reports' folder next to the project file with Port Diff Reports.

Note: IP cores older than Quartus Prime software version 12.0 do not support upgrade. Altera verifies that the current version of the Quartus Prime 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

Upgrading IP at Command-Line

You can upgrade an IP core at the command-line if the IP core supports auto upgrade. IP cores that do not support automatic upgrade do not support command-line upgrade.

- To upgrade a single IP core at the command-line, type the following command:

```
quartus_sh –ip_upgrade –variation_files <my_ip>.<qsys,.v, .vhd> <qii_project>

Example:
quartus_sh -ip_upgrade -variation_files mega/pll25.qsys hps_testx
```

- To simultaneously upgrade multiple IP cores at the command-line, type the following command:

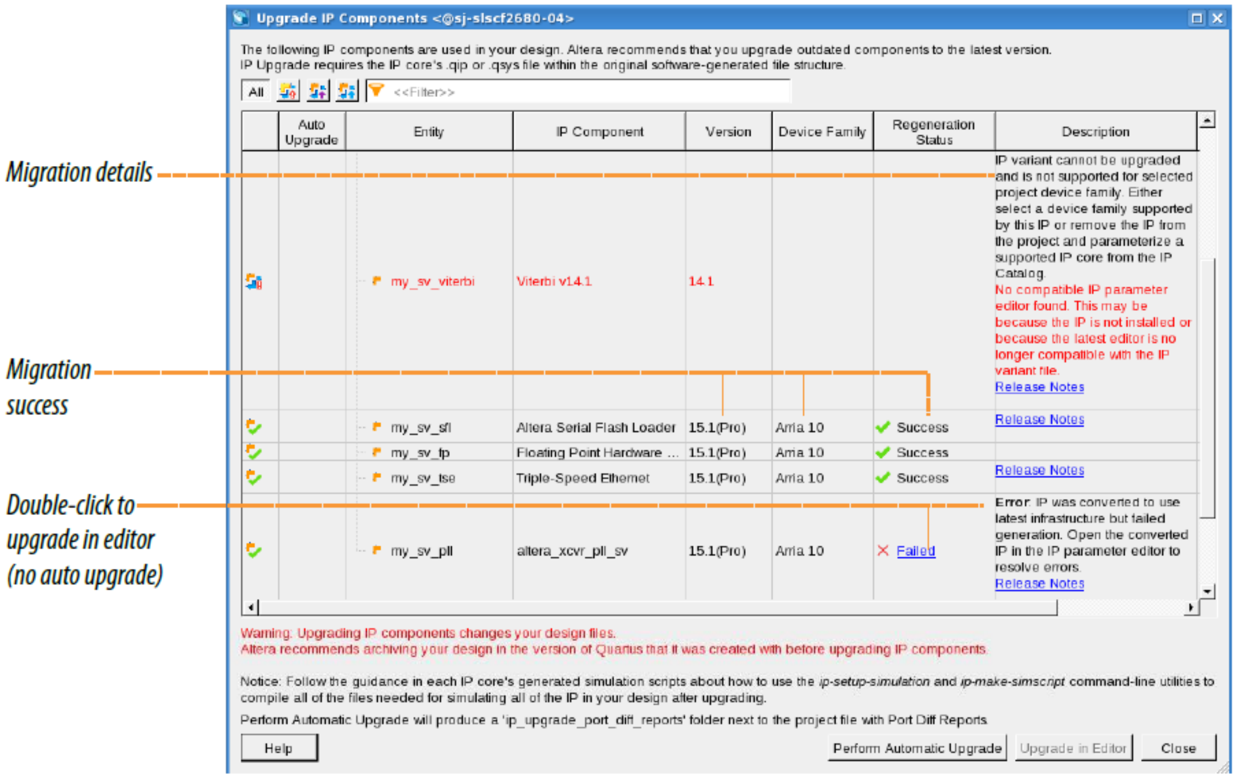
```
quartus_sh –ip_upgrade –variation_files “<my_ip1>.<qsys,.v, .vhd>>;
<my_ip_filepath/my_ip2>.<hdl>”  <qii_project>

Example:
quartus_sh -ip_upgrade -variation_files "mega/pll_tx2.qsys;mega/pll3.qsys"
hps_testx
```

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. Most Altera IP cores support automatic migration. Some IP cores require manual IP regeneration for migration. Some IP cores 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.

Figure 2-21: IP Core Device Migration


Migration details
Migration success
Double-click to upgrade in editor (no auto upgrade)
Upgrade IP Components <@sj-slscf2680-04>
All
<<Filter>>

The following IP components are used in your design. Altera recommends that you upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original software-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description
		my_sv_viterbi	Viterbi v14.1	14.1			IP variant cannot be upgraded and is not supported for selected project device family. Either select a device family supported by this IP or remove the IP from the project and parameterize a supported IP core from the IP Catalog. No compatible IP parameter editor found. This may be because the IP is not installed or because the latest editor is no longer compatible with the IP variant file. Release Notes
		my_sv_sfl	Altera Serial Flash Loader	15.1(Pro)	Arria 10	✓ Success	Release Notes
		my_sv_fp	Floating Point Hardware ...	15.1(Pro)	Arria 10	✓ Success	
		my_sv_tse	Triple-Speed Ethernet	15.1(Pro)	Arria 10	✓ Success	Release Notes
		my_sv_pll	altera_xcvr_pll_sv	15.1(Pro)	Arria 10	✗ Failed	Error: IP was converted to use latest infrastructure but failed generation. Open the converted IP in the IP parameter editor to resolve errors. Release Notes

Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design in the version of Quartus that it was created with before upgrading IP components.

Notice: Follow the guidance in each IP core's generated simulation scripts about how to use the *ip-setup-simulation* and *ip-make-simscript* command-line utilities to compile all of the files needed for simulating all of the IP in your design after upgrading.

Perform Automatic Upgrade will produce a 'ip_upgrade_port_diff_reports' folder next to the project file with Port Diff Reports.

1. Click **File** > **Open Project** and open the Quartus Prime project containing IP for migration to another device in the original version of the Quartus Prime software. If prompted, click **Yes** to change to a supported device family.
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. If the parameter editor specifies a **Currently selected device family**, turn off **Match project/default**, and then select the new target device family.
6. Click **Generate HDL**, and then confirm the **Synthesis** and **Simulation** file options. Verilog HDL is the default output file format specified. If your original IP core was generated for VHDL, select **VHDL** to retain the original output format.
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

Troubleshooting IP or Qsys System Upgrade

The **Upgrade IP Components** dialog box reports the version and status of each IP core and Qsys system following upgrade or migration. If any upgrade or migration fails, the **Upgrade IP Components** dialog box provides information to help you resolve any errors.

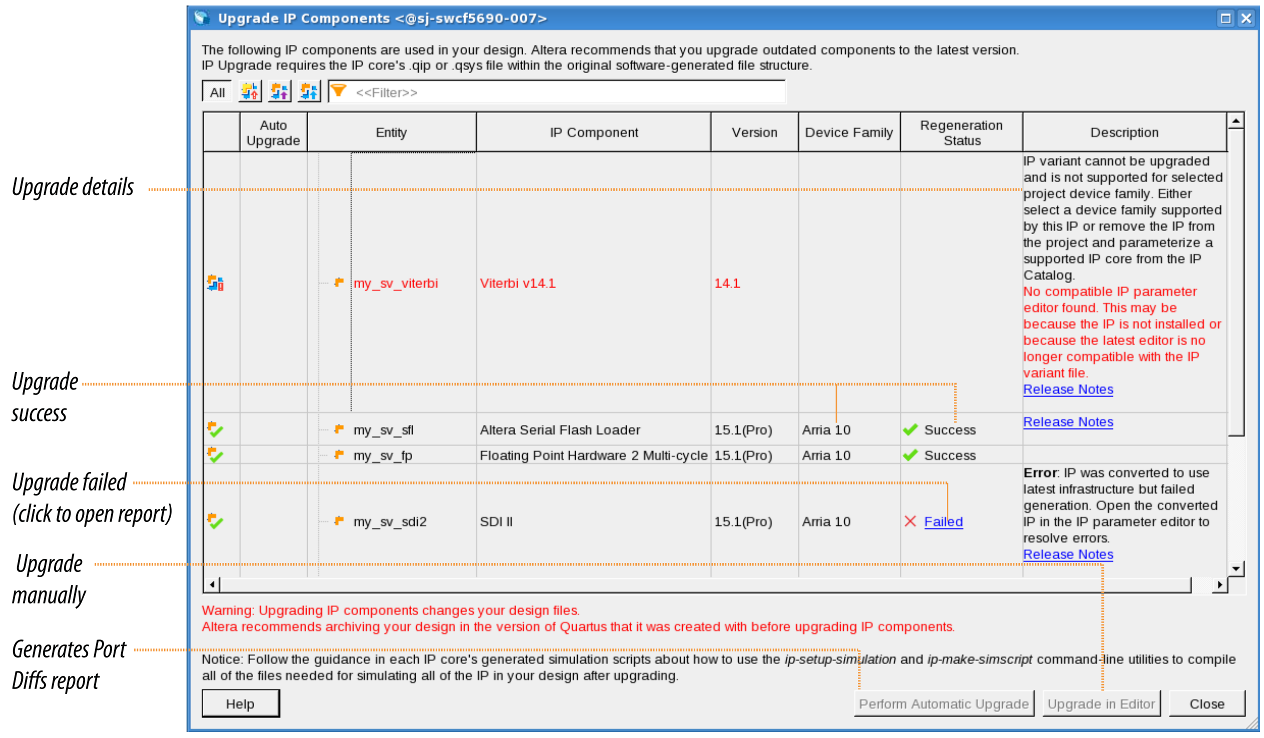
Note: Make sure that your IP variation names or paths do not include spaces. Spaces can be problematic for IP generation.

During automatic or manual upgrade, the Messages window dynamically displays upgrade information for each IP core or Qsys system. You can use the following information to help you resolve any upgrade errors following upgrade or migration.

Table 2-8: IP Upgrade Error Information

Upgrade IP Components Field	Description
Regneration Status	Displays the "Success" or "Failed" status of each upgrade or migration. Click the status of any failed upgrade to open a detailed **IP Upgrade Report**.
Version	Dynamically updates to the new version number when upgrade is successful. The text is red when upgrade is required.
Device Family	Dynamically updates to the new device family when migration is successful. The text is red when upgrade is required.
Description	Summarizes IP release information and displays actionable, corrective action for resolving upgrade or migration failures. Follow these instructions to resolve upgrade failures. Click the **Release Notes** link for the latest known issues about the Altera IP core.
Perform Automatic Upgrade	Runs automatic upgrade on all IP cores that support auto upgrade. Also, automatically generates a *<Project Directory>*/**ip_upgrade_port_diff_report** report for IP cores or Qsys systems that fail upgrade. Review these reports to determine any port differences between the current and previous IP core version.

Figure 2-22: Resolving Upgrade Errors


Upgrade details
Upgrade success
Upgrade failed (click to open report)
Upgrade manually
Generates Port Diffs report
All
<<Filter>>
Help
Perform Automatic Upgrade
Upgrade in Editor
Close

Upgrade IP Components <@sj-swcf5690-007>

The following IP components are used in your design. Altera recommends that you upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original software-generated file structure.

Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description
	my_sv_viterbi	Viterbi v14.1	14.1			IP variant cannot be upgraded and is not supported for selected project device family. Either select a device family supported by this IP or remove the IP from the project and parameterize a supported IP core from the IP Catalog. No compatible IP parameter editor found. This may be because the IP is not installed or because the latest editor is no longer compatible with the IP variant file. Release Notes
	my_sv_sfl	Altera Serial Flash Loader	15.1(Pro)	Arria 10	✓ Success	Release Notes
	my_sv_fp	Floating Point Hardware 2 Multi-cycle	15.1(Pro)	Arria 10	✓ Success	
	my_sv_sdi2	SDI II	15.1(Pro)	Arria 10	✗ Failed	**Error**: IP was converted to use latest infrastructure but failed generation. Open the converted IP in the IP parameter editor to resolve errors. Release Notes

Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design in the version of Quartus that it was created with before upgrading IP components.

Notice: Follow the guidance in each IP core's generated simulation scripts about how to use the *ip-setup-simulation* and *ip-make-simscript* command-line utilities to compile all of the files needed for simulating all of the IP in your design after upgrading.

Use the following techniques to resolve errors if your Altera IP core or Qsys system "Failed" to upgrade versions or migrate to another device. Review and implement the instructions in the **Description** field, including one or more of the following:

1. If the IP variant is not supported in the current version of the software, right-click the component and click **Remove IP Component from Project**. Replace this IP core or Qsys system with one supported in the current version of the software.
2. If the IP variant is not supported by the current target device, select a supported device family for the project, or replace the IP variant with a suitable replacement that supports your target device.
3. If an upgrade or migration fails, click **Failed** in the **Regeneration Status** field to display and review details of the **IP Upgrade Report**. Click the **Release Notes** link for the latest known issues about the Altera IP core. Use this information to determine the nature of the upgrade or migration failure and make corrections before upgrade.

Figure 2-23: IP Upgrade Report

```
IP Upgrade report for a10_ip_upgrade
Tue Aug 25 13:30:53 2015
Quartus Prime Version 15.1.0 Build 166 08/23/2015 SJ Pro Edition


---------------------
; Table of Contents ;
---------------------
  1. Legal Notice
  2. IP Upgrade Summary
  3. Successfully Upgraded IP Components
  4. Failed Upgrade IP Components
  5. IP Upgrade Messages



----------------
; Legal Notice ;
----------------
Copyright (C) 1991-2015 Altera Corporation. All rights reserved.
Your use of Altera Corporation's design tools, logic functions
and other software and tools, and its AMPP partner logic
functions, and any output files from any of the foregoing
(including device programming or simulation files), and any
associated documentation or information are expressly subject
to the terms and conditions of the Altera Program License
Subscription Agreement, the Altera Quartus Prime License Agreement,
the Altera MegaCore Function License Agreement, or other
applicable license agreement, including, without limitation,
that your use is for the sole purpose of programming logic
devices manufactured by Altera and sold by Altera or its
authorized distributors.  Please refer to the applicable
agreement for further details.



+----------------------------------------------------------------------------+
; IP Upgrade Summary                                                         ;
+-------------------------------+--------------------------------------------+
; IP Components Upgrade Status  ; Passed - Tue Aug 25 13:30:53 2015          ;
; Quartus Prime Version         ; 15.1.0 Build 166 08/23/2015 SJ Pro Edition ;
; Revision Name                 ; a10_ip_upgrade                             ;
; Top-level Entity Name         ; a10_ip_upgrade                             ;
; Family                        ; Arria 10                                   ;
+-------------------------------+--------------------------------------------+
```

4. Run **Perform Automatic Upgrade** to automatically generate an **IP Ports Diff** report for each IP core or Qsys system that fails upgrade. Review the reports to determine any port differences between the

current and previous IP core version. Then, click **Upgrade in Editor** to make specific port changes and regenerate your IP core or Qsys system.

5. If your IP core or Qsys system does not support **Perform Automatic Upgrade**, click **Upgrade in Editor** to resolve errors and regenerate the component in the parameter editor.

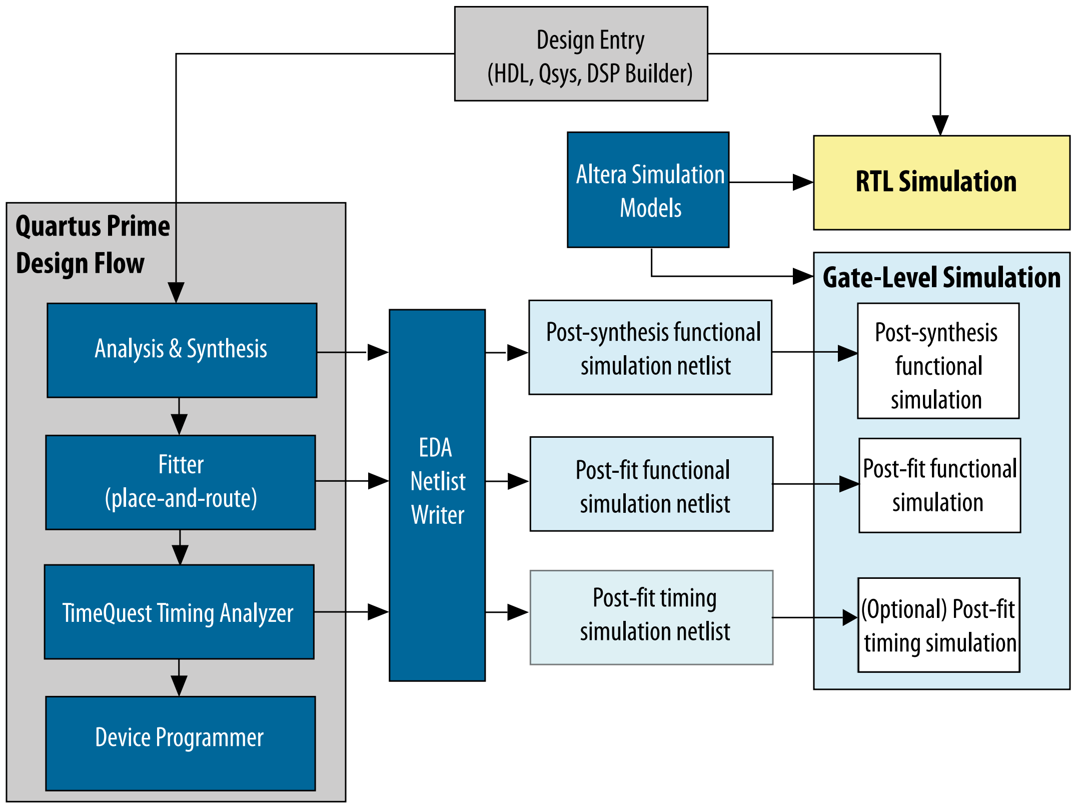
Simulating Altera IP Cores in other EDA Tools

The Quartus Prime 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 `ip-setup-simulation` utility to generate a unified, version-agnostic IP simulation script for all Altera IP cores in your design. You can incorporate the IP simulation scripts into your top-level script.

Figure 2-24: Simulation in Quartus Prime Design Flow


Design Entry (HDL, Qsys, DSP Builder)
Altera Simulation Models
RTL Simulation
Quartus Prime 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: 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 use Altera-provided utilities to generate a combined simulation script for Altera IP cores in your design. You can source these IP simulation scripts in your top-level project script. You can modify and reuse these simulation scripts to fit your simulation requirements.

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 Version-Independent IP and Qsys Simulation Scripts

The Quartus Prime software includes useful utilities that generate simulation scripts for each IP core or Qsys system in your design. You can use these utilities to produce a single simulation script that does not require manual update for upgrades to Quartus Prime software or IP versions.

This scripted method generates simulation scripts that support ModelSim-Altera, all supported versions of Questa-SIM, VCS, VCSMX, NCSim, and Aldec simulators. These generated scripts are not suitable for entire design simulation because they lack top-level design information. However, you can easily source the generated scripts from your top-level simulation script. You can incorporate templates from the generated scripts into a top-level script.

Use the `ip-setup-simulation` utility to find all Altera IP cores and Qsys systems in your project. Next, run the `ip-make-simscript` utility to generate a combined IP simulation script. The `ip-setup-simulation` utility also automates regeneration of a combined simulation script following upgrade of the software. If you use simulation scripts, run the `ip-setup-simulation` utility after upgrading software or IP core version.

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.

Table 2-9: IP Simulation Script Utilities

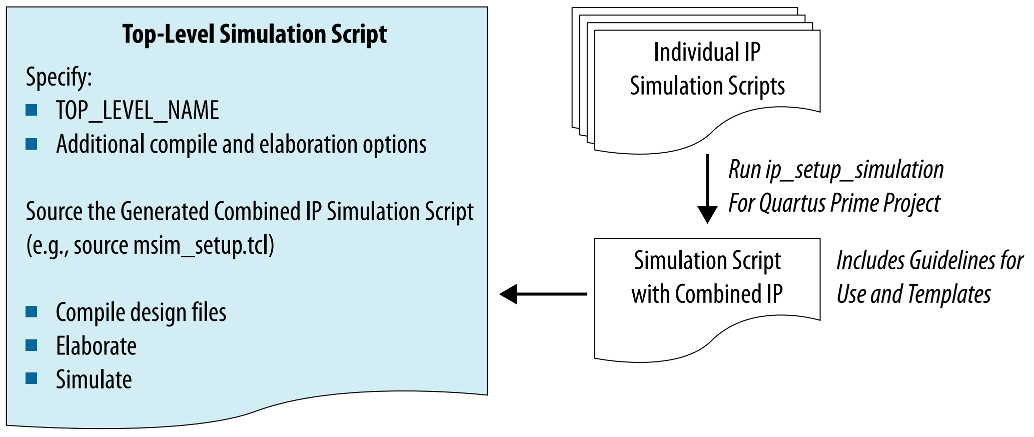
Utility	Description	Syntax	Generated Files
`ip-setup-simulation`	Finds all Altera IP cores in your project and automates regeneration of a combined simulation script after upgrading software or IP versions.	`ip-setup-simulation --quartus-project=<project>.qpf --output-directory=<directory>`	N/A

Utility	Description	Syntax	Generated Files
`ip-make-simscript`	Generates a single, combined simulation script for all of the IP cores specified on the command line. To use `ip-make-simscript`, specify one or more **.spd** files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries. Use the `compileto- work` option to compile all simulation files into a single work library. Use the `--use-relative-paths` option to use relative paths whenever possible	`ip-make-simscript --spd=<`*`ipA.spd,ipB.spd`*`> --output-directory=<directory>`	• Aldec—**aldec/rivierapro_setup.tcl** • Cadence—**cadence/ncsim_setup.sh** • Mentor Graphics—**mentor/msim_setup.tcl** • Synopsys—**synopsys/vcs/vcs_setup.sh**

Incorporating IP Simulation Scripts in Top-Level Scripts

You can incorporate generated IP core simulation scripts into a top-level simulation script that controls simulation of your entire design. After generating a combined IP simulation script, you can copy the template sections and modify them for use in a new top-level script file.

Figure 2-25: Incorporating IP Simulation Scripts into Top-Level Script


Top-Level Simulation Script
Specify:
TOP_LEVEL_NAME
Additional compile and elaboration options
Source the Generated Combined IP Simulation Script (e.g., source msim_setup.tcl)
Compile design files
Elaborate
Simulate
Individual IP Simulation Scripts
Run ip_setup_simulation For Quartus Prime Project
Simulation Script with Combined IP
Includes Guidelines for Use and Templates

1. Run `ip-setup-simulation` on the project:

```
ip-setup-simulation --quartus-project=<project>.qpf
    --output-directory=<directory>
```

2. Copy the template sections from the simulator-specific generated scripts and paste them into a new top-level file. The examples in this document assume that the top-level simulation script file is in the

project directory, and that the generated simulation scripts are located in a directory one level below the project directory.

3. After pasting the template sections, remove the comments at the beginning of each line from the copied template sections.
4. Make any other modification to match your design simulation requirements, for example:
   a. Specify the `TOP_LEVEL_NAME` variable to the design's simulation top-level file.
   b. Compile the top-level HDL file (e.g. a test program) and all other files in the design.
   c. Specify any other changes, such as using the `grep` command-line utility to search a transcript file for error signatures, or e-mail a report.

Incorporating Aldec IP Simulation Scripts

To incorporate generated Aldec simulation scripts into a top-level project simulation script, follow these steps:

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, **sim_top.tcl**.

```
# # Start of template
# # If the copied and modified template file is "aldec.do", run it as:
# # vsim -c -do aldec.do
# #
# # Source the generated sim script
# source rivierapro_setup.tcl
# # Compile eda/sim_lib contents first
# dev_com
# # Override the top-level name (so that elab is useful)
# set TOP_LEVEL_NAME top
# # Compile the standalone IP.
# com
# # Compile the user top-level
# vlog -sv2k5 ../../top.sv
# # Elaborate the design.
# elab
# # Run the simulation
# run
# # Report success to the shell
# exit -code 0
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "aldec.do", run it as:
# vsim -c -do aldec.do
#
# Source the generated sim script source rivierapro_setup.tcl
# Compile eda/sim_lib contents first dev_com
# Override the top-level name (so that elab is useful)
set TOP_LEVEL_NAME top
# Compile the standalone IP.
com
# Compile the user top-level vlog -sv2k5 ../../top.sv
# Elaborate the design.
elab
# Run the simulation
run
# Report success to the shell
```

```
exit -code 0
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
set TOP_LEVEL_NAME sim_top vlog –sv2k5 ../../sim_top.sv
```

4. Specify any other changes required to match your design simulation requirements.
5. Run the new top-level script from the generated simulation directory:

```
vsim –c –do <path to sim_top>.tcl
```

Incorporating Cadence IP Simulation Scripts

To incorporate generated Cadence IP simulation scripts into a top-level project simulation script, follow these steps:

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, **ncsim.sh**.

```
# # Start of template
# # If the copied and modified template file is "ncsim.sh", run it as:
# # ./ncsim.sh
# #
# # Do the file copy, dev_com and com steps
# source ncsim_setup.sh \
# SKIP_ELAB=1 \
# SKIP_SIM=1
#
# # Compile the top level module
# ncvlog -sv "$QSYS_SIMDIR/../top.sv"
#
# # Do the elaboration and sim steps
# # Override the top-level name
# # Override the user-defined sim options, so the simulation
# # runs forever (until $finish()).
# source ncsim_setup.sh \
# SKIP_FILE_COPY=1 \
# SKIP_DEV_COM=1 \
# SKIP_COM=1 \
# TOP_LEVEL_NAME=top \
# USER_DEFINED_SIM_OPTIONS=""
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "ncsim.sh", run it as:
# ./ncsim.sh
#
# Do the file copy, dev_com and com steps
source ncsim_setup.sh \
SKIP_ELAB=1 \
SKIP_SIM=1
# Compile the top level module
ncvlog -sv "$QSYS_SIMDIR/../top.sv"
# Do the elaboration and sim steps
# Override the top-level name
# Override the user-defined sim options, so the simulation
# runs forever (until $finish()).
source ncsim_setup.sh \
SKIP_FILE_COPY=1 \
SKIP_DEV_COM=1 \
SKIP_COM=1 \
TOP_LEVEL_NAME=top \
USER_DEFINED_SIM_OPTIONS=""
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
TOP_LEVEL_NAME=sim_top \
```

4. Make the appropriate changes to the compilation of the your top-level file, for example:

```
ncvlog -sv "$QSYS_SIMDIR/../top.sv"
```

5. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **ncsim.sh**.

Incorporating ModelSim IP Simulation Scripts

To incorporate generated ModelSim IP simulation scripts into a top-level project simulation script, follow these steps:

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, **sim_top.tcl**.

```
# # Start of template
# # If the copied and modified template file is "mentor.do", run it
# # as: vsim -c -do mentor.do
# #
# # Source the generated sim script
# source msim_setup.tcl
# # Compile eda/sim_lib contents first
# dev_com
# # Override the top-level name (so that elab is useful)
# set TOP_LEVEL_NAME top
# # Compile the standalone IP.
# com
# # Compile the user top-level
# vlog -sv ../../top.sv
# # Elaborate the design.
# elab
# # Run the simulation
# run -a
# # Report success to the shell
# exit -code 0
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "mentor.do", run it
# as: vsim -c -do mentor.do
#
# Source the generated sim script source msim_setup.tcl
# Compile eda/sim_lib contents first
dev_com
# Override the top-level name (so that elab is useful)
set TOP_LEVEL_NAME top
# Compile the standalone IP.
com
# Compile the user top-level vlog -sv ../../top.sv
# Elaborate the design.
elab
```

```
# Run the simulation
run -a
# Report success to the shell
exit -code 0
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

   ```
   set TOP_LEVEL_NAME sim_top vlog -sv ../../sim_top.sv
   ```

4. Specify any other changes required to match your design simulation requirements.
5. Run the resulting top-level script from the generated simulation directory:

   ```
   vsim –c –do <path to sim_top>.tcl
   ```

Incorporating VCS IP Simulation Scripts

To incorporate generated Synopsys VCS simulation scripts into a top-level project simulation script, follow these steps:

1. The generated simulation script contains these template lines. Cut and paste the lines preceding the "helper file" into a new executable file. For example, **synopsys_vcs.f**.

   ```
   # # Start of template
   # # If the copied and modified template file is "vcs_sim.sh", run it
   # # as: ./vcs_sim.sh
   # #
   # # Override the top-level name
   # # specify a command file containing elaboration options
   # # (system verilog extension, and compile the top-level).
   # # Override the user-defined sim options, so the simulation
   # # runs forever (until $finish()).
   # source vcs_setup.sh \
   # TOP_LEVEL_NAME=top \
   # USER_DEFINED_ELAB_OPTIONS="'-f ../../../synopsys_vcs.f'" \
   # USER_DEFINED_SIM_OPTIONS=""
   #
   # # helper file: synopsys_vcs.f
   # +systemverilogext+.sv
   # ../../../top.sv
   # # End of template
   ```

2. Delete the first two characters of each line (comment and space) for the **vcs.sh** file, as shown below:

   ```
   # Start of template
   # If the copied and modified template file is "vcs_sim.sh", run it
   # as: ./vcs_sim.sh
   #
   # Override the top-level name
   # specify a command file containing elaboration options
   # (system verilog extension, and compile the top-level).
   # Override the user-defined sim options, so the simulation
   # runs forever (until $finish()).
   source vcs_setup.sh \
   TOP_LEVEL_NAME=top \
   ```

```
USER_DEFINED_ELAB_OPTIONS="'-f ../../../synopsys_vcs.f'" \
USER_DEFINED_SIM_OPTIONS=""
```

3. Delete the first two characters of each line (comment and space) for the **synopsys_vcs.f** file, as shown below:

```
# helper file: synopsys_vcs.f
 +systemverilogext+.sv
 ../../../top.sv
# End of template
```

4. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
TOP_LEVEL_NAME=sim_top \
```

5. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **vcs_sim.sh**.

Incorporating VCS MX IP Simulation Scripts

To incorporate generated Synopsys VCS MX simulation scripts for use in top-level project simulation scripts, follow these steps:

1. The generated simulation script contains these template lines. Cut and paste the lines preceding the "helper file" into a new executable file. For example, **vcsmx.sh**.

```
# # Start of template
# # If the copied and modified template file is "vcsmx_sim.sh", run
# # it as: ./vcsmx_sim.sh
# #
# # Do the file copy, dev_com and com steps
# source vcsmx_setup.sh \
# SKIP_ELAB=1 \

# SKIP_SIM=1
#
# # Compile the top level module vlogan +v2k
     +systemverilogext+.sv "$QSYS_SIMDIR/../top.sv"

# # Do the elaboration and sim steps
# # Override the top-level name
# # Override the user-defined sim options, so the simulation runs
# # forever (until $finish()).
# source vcsmx_setup.sh \
# SKIP_FILE_COPY=1 \
# SKIP_DEV_COM=1 \
# SKIP_COM=1 \
# TOP_LEVEL_NAME="'-top top'" \
# USER_DEFINED_SIM_OPTIONS=""
# # End of template
```

2. Delete the first two characters of each line (comment and space), as shown below:

```
# Start of template
# If the copied and modified template file is "vcsmx_sim.sh", run
# it as: ./vcsmx_sim.sh
#
# Do the file copy, dev_com and com steps
source vcsmx_setup.sh \
SKIP_ELAB=1 \
SKIP_SIM=1
```

```
# Compile the top level module
vlogan +v2k +systemverilogext+.sv "$QSYS_SIMDIR/../top.sv"

# Do the elaboration and sim steps
# Override the top-level name
# Override the user-defined sim options, so the simulation runs
# forever (until $finish()).
source vcsmx_setup.sh \
SKIP_FILE_COPY=1 \
SKIP_DEV_COM=1 \
SKIP_COM=1 \
TOP_LEVEL_NAME="'-top top'" \
USER_DEFINED_SIM_OPTIONS=""
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

   ```
   TOP_LEVEL_NAME=”-top sim_top’” \
   ```

4. Make the appropriate changes to the compilation of the your top-level file, for example:

   ```
   vlogan +v2k +systemverilogext+.sv "$QSYS_SIMDIR/../sim_top.sv"
   ```

5. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **vcsmx_sim.sh**.

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 Prime 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 Prime project.

Related Information

Quartus Prime 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;
```

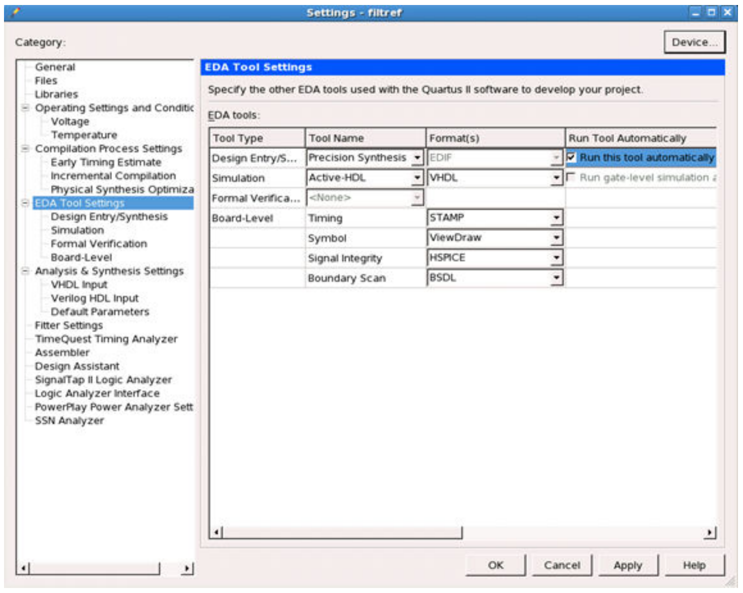
Integrating Other EDA Tools

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

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

- 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 2-26: 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
Formal Verifica...	<None>		
Board-Level	Timing	STAMP	
	Symbol	ViewDraw	
	Signal Integrity	HSPICE	
	Boundary Scan	BSDL	

Related Information

Mentor Graphics Precision Synthesis SupportGraphics on page 16-1

Simulating Altera Designs

Managing Team-based Projects

The Quartus Prime 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
- Archiving Projects on page 2-45
- Using External Revision Control on page 2-46
- Migrating Projects Across Operating Systems on page 2-46

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 Prime 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.

Archiving Projects

You can save the elements of a project in a single, compressed Quartus Prime 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 Prime 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 Prime 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**.

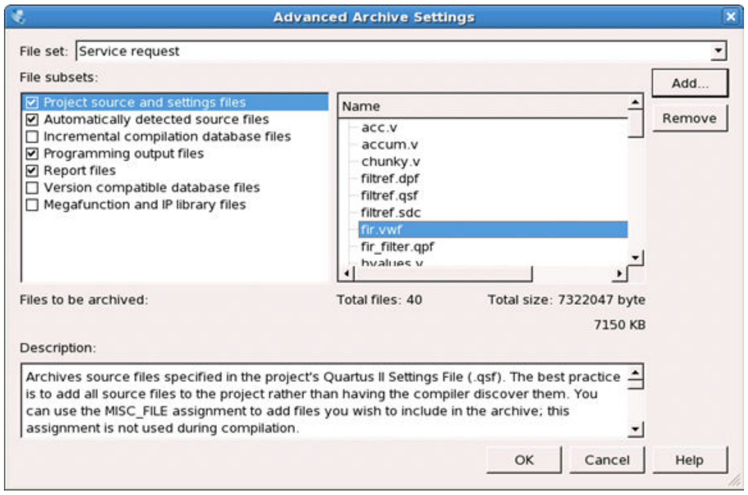
Archiving Projects for Altera Service Requests

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 2-27: Archiving Project for Service Request



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 Prime 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

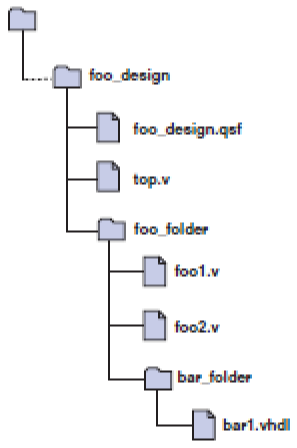
Consider the following cross-platform issues when moving your project from one operating system to another (for example, from Windows to Linux).

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 Prime 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 2-28: All Inclusive Project Directory Structure

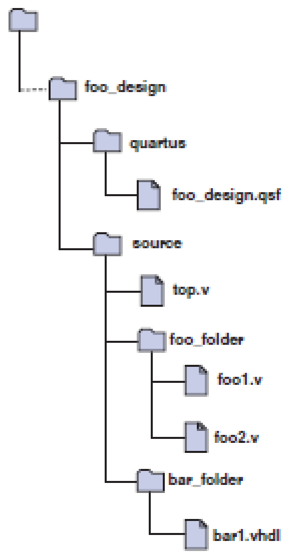


Use Relative Paths

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 2-29: Quartus Prime 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 Prime 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 Prime 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 Prime 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 2-49

Set Current Revision Command on page 2-49

Get Project Revisions Command on page 2-49

Delete Revision Command on page 2-49

Create Revision Command

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

Option	Description
`based_on` (optional)	Specifies the revision name on which the new revision bases its settings.
`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

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 Library Commands

Use the following commands to script project library changes.

Specify Project Libraries With SEARCH_PATH Assignment on page 2-50

Report Specified Project Libraries Commands on page 2-50

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 2-10: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Added Generating Version-Independent IP Simulation Scripts topic. • Added example IP simulation script templates for supported simulators. • Added Incorporating IP Simulation Scripts in Top-Level Scripts topic. • Added Troubleshooting IP Upgrade topic. • Updated IP Catalog and parameter editor descriptions for GUI changes. • Updated IP upgrade and migration steps for latest GUI changes. • Updated Generating IP Cores process for GUI changes. • Updated Files Generated for IP Cores and Qsys system description. • Removed references to devices and features not supported in version 15.1. • Changed instances of *Quartus II* to *Quartus Prime*.
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.

Date	Version	Changes
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 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.
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 Handbook Archive

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

Design Planning with the Quartus Prime Software 3

2015.11.02

QPP5V1 Subscribe Send Feedback

Design Planning with the Quartus Prime Software

Platform planning—the early feasibility analysis of physical constraints—is a fundamental early step in advanced FPGA design. FPGA device densities and complex are increasing and designs oftem 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.

Note: The BluePrint Platform Designer helps you to accurately plan constraints for design implementation. Use BluePrint to prototype interface implementations and rapidly define a legal device floorplan for Arria 10 devices.

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 Prime 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.

Related Information

- BluePrint Design Planning

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 3-6
  For guidelines related to analyzing and debugging the device after it is in the system.
- **Planning for Hierarchical and Team-Based Design**
  For more suggestions on team-based designs.
- **Using Qsys and Standard Interfaces in System Design** on page 3-2
  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 Prime 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 Prime 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 Prime 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 Prime 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 Prime software, which can significantly affect the resource utilization and timing performance of your design.

Related Information

- **Planning for On-Chip Debugging Tools** on page 3-6
  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 Prime 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 Prime 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

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 Prime 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 Prime software performs voltage compatibility checks of those pins during 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime Help.
- **PowerPlay Early Power Estimator and Power Analyzer**
  The PowerPlay EPE spreadsheets for each supported device family are available on the Altera website.

Selecting Third-Party EDA Tools

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

Synthesis Tool

The Quartus Prime 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 Prime 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 Prime software to get accurate timing analysis and logic utilization results.

The synthesis tool you choose may allow you to create a Quartus Prime 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 Prime 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.

Simulation Tool

Altera provides the Mentor Graphics ModelSim®-Altera Starter Edition with the Quartus Prime 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 Prime software can generate both functional and timing netlist files for ModelSim and other third-party simulators.

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

Formal Verification Tools

Consider whether the Quartus Prime 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.

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 Prime 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 Prime 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.

Table 3-1: 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.

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

Recommended Design Practices on page 10-1
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 11-1
  For HDL coding examples and recommendations, refer to the Recommended HDL Coding Styles chapter in volume 1 of the Quartus Prime 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 Prime 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 Prime 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 Prime Software** on page 13-1
  For information about metastability analysis, reporting, and optimization features in the Quartus Prime software

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 Prime 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.

Related Information

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

Document Revision History

Table 3-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Added references to BluePrint Design Planning chapter. • Changed instances of *Quartus II* to *Quartus Prime*.
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 Prime 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 Prime Language Templates

Related Information

Quartus Handbook Archive

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

Creating a System With Qsys

2015.11.02

QPP5V1 Subscribe Send Feedback

Qsys is a system integration tool included as part of the Quartus Prime software. Qsys simplifies the task of defining and integrating customized IP Components (IP Cores) into your designs.

Qsys facilitates design reuse by packaging and integrating your custom IP components with Altera and third-party IP components. Qsys automatically creates interconnect logic from the high-level connectivity that you specify, which eliminates the error-prone and time-consuming task of writing HDL to specify system-level connections.

Qsys is a more powerful tool if you design your custom IP components using standard interfaces available in the Qsys IP Catalog. Standard interfaces inter-operate efficiently with the Altera IP components, and you can take advantage of bus functional models (BFMs), monitors, and other verification IP to verify your systems.

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 Prime 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 Prime software

Note: For information on how to define and generate stand-alone IP cores for use in your Quartus Prime software projects, refer to *Introduction to Altera IP Cores* and *Managing Quartus Prime Projects*.

Related Information

- **Introduction to Altera IP Cores**
- **Managing Quartus Prime Projects** on page 2-1
- **Avalon Interface Specifications**
- **AMBA Protocol Specifications**

Interface Support in Qsys

IP components (IP Cores) 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 4-1: 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.

Introduction to the Qsys IP Catalog

The Qsys IP Catalog offers a broad range of configurable IP Cores optimized for Altera devices to use in your Qsys designs.

The Quartus Prime 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. The IP Catalog can include Altera-provided IP components, third-party IP components, custom IP components that you create in the Qsys Component Editor, and previously generated Qsys systems.

The Qsys IP Catalog includes the following IP component types:

- Microprocessors, such as the Nios® II processor
- DSP IP cores, 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 cores, such as the Avalon-ST Multiplexer
- Qsys Interconnect
- Verification IP (VIP) Bus Functional Models (BFMs)

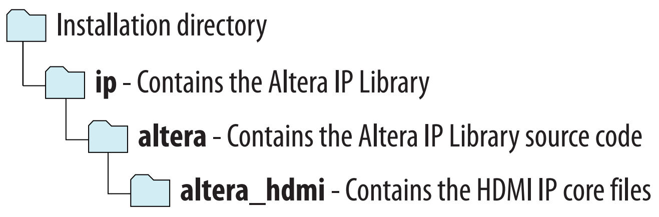
Related Information

- **Introduction to Altera IP Cores**

Installing and Licensing IP Cores

The Quartus Prime software includes the Altera IP Library. The library provides many useful IP core functions for production use without additional license. You can fully evaluate any licensed Altera IP core in simulation and in hardware until you are satisfied with its functionality and performance. The HDMI IP core is part of the Altera MegaCore IP Library, which is distributed with the Quartus Prime software and downloadable from the Altera web site.

Figure 4-1: HDMI Installation Path


Installation directory
ip - Contains the Altera IP Library
altera - Contains the Altera IP Library source code
altera_hdmi - Contains the HDMI IP core files

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

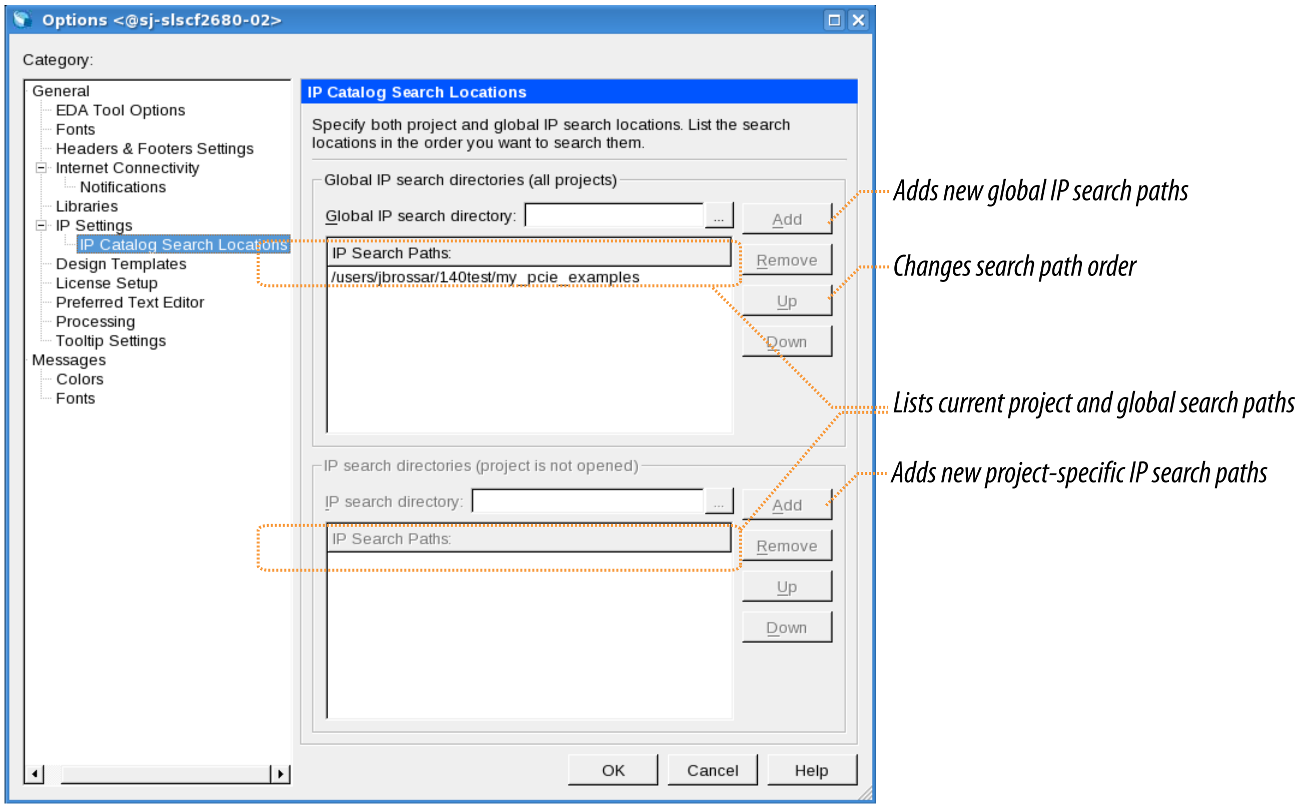
After you purchase a license for the HDMI IP core, you can request a license file from the Altera's licensing site and install it on your computer. When you request a license file, Altera emails you a **license.dat file**. If you do not have Internet access, contact your local Altera representative.

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.

Figure 4-2: Specifying IP Search Locations


Options <@sj-slscf2680-02>
Category:
General
EDA Tool Options
Fonts
Headers & Footers Settings
Internet Connectivity
Notifications
Libraries
IP Settings
IP Catalog Search Locations
Design Templates
License Setup
Preferred Text Editor
Processing
Tooltip Settings
Messages
Colors
Fonts
IP Catalog Search Locations
Global IP search directories (all projects)
Global IP search directory:
Add
Remove
Up
Down
IP Search Paths:
IP search directories (project is not opened)
IP search directory:
IP Search Paths:
OK
Cancel
Help
Adds new global IP search paths
Changes search path order
Lists current project and global search paths
Adds new project-specific IP search paths

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

/users/jbrossar/140test/my_pcie_examples

The Quartus Prime 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 Prime 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 4-2: IP Search Locations

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

If the Quartus Prime 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 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 update the IP Catalog by clicking **Refresh IP Catalog** in the drop-down list. In Qsys, click **File** > **Refresh System** to update the IP Catalog.

General Settings for IP

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

Table 4-3: 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 Prime 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 regenerate 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 Prime software searches for IP cores in the project directory, in the Altera installation directory, and in the IP search path.

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 Prime 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 4-1: 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 4-2: 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 4-78

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

Create a Qsys System

Click **Tools** > **Qsys** in the Quartus Prime software to open Qsys. A **.qsys** or **.qip** file represents your Qsys system in your Quartus Prime software project.

Related Information

- Creating Qsys Components on page 5-1
- Component Interface Tcl Reference on page 8-1

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 Prime 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 Prime project device settings.

The Quartus Prime software always uses the device specified in the Quartus Prime 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 Prime project settings. Also, when you open Qsys from within the Quartus Prime software, the Quartus Prime project device settings apply.

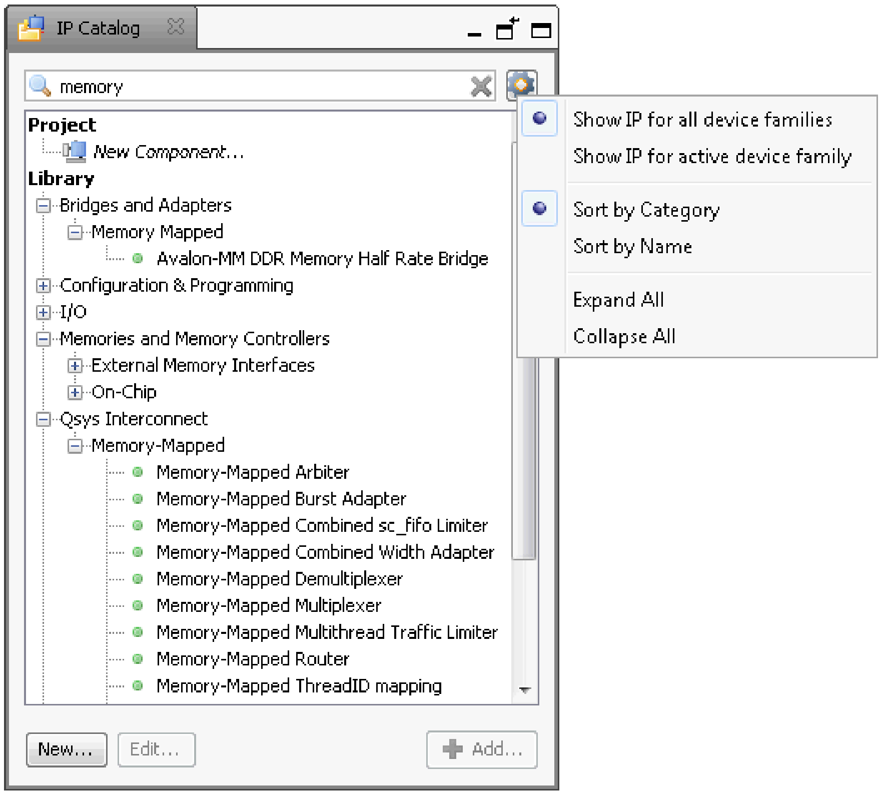
Add IP Components (IP Cores) to a Qsys System

The Qsys IP Catalog displays IP components (IP Cores) available for your target device.

Double-click any component in the IP Catalog to launch the parameter editor. The parameter editor allows you to create a custom IP component 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.

1. Right-click any IP component name in the Qsys IP Catalog to display details about device support, installation location, versions, and links to documentation.
2. To locate a specific type of component, type some or all of the component's name in the **IP Catalog** search box.
   For example, 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.
3. Double-click any component to launch the parameter editor.
   The parameter editor opens where you can set parameter values, and view the block diagram for component.
4. For IP components that have preset parameter values, select the prest file in the preset editor, and then click Apply.
   Allows you to instantly apply preset parameter values for the IP component appropriate for a specific application.
5. Click **Finish** to complete customization of the IP component.
   The IP component appears in the **System Contents** tab.

Figure 4-3: Qsys 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.

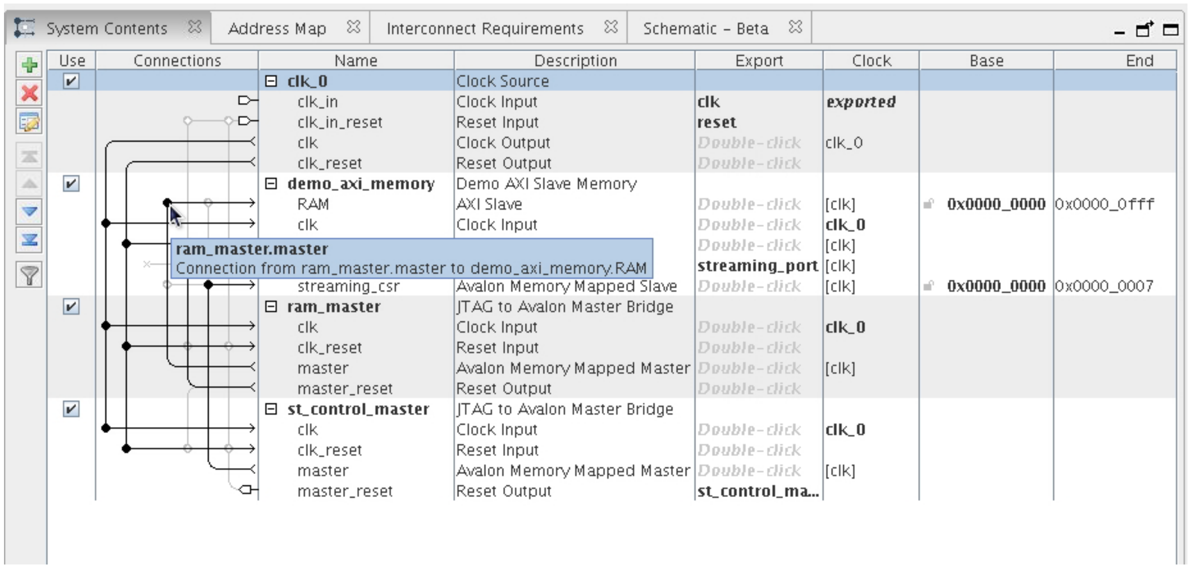
Possible connections between interfaces appear as gray lines and open circles. To make a connection, click the open circle at the intersection of the interfaces. When you make a connection, Qsys draws the connection line in black and fills the connection circle. Clicking a filled-in circle removes a 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, the **Clock Crossing**, **Data Width**, and **Burst** columns provide interconnect information about added adapters that can sometimes result in slower f_{Max}, or larger area.

Figure 4-4: 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 allows you to focus on a selected element in your system under different perspectives.

The 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.

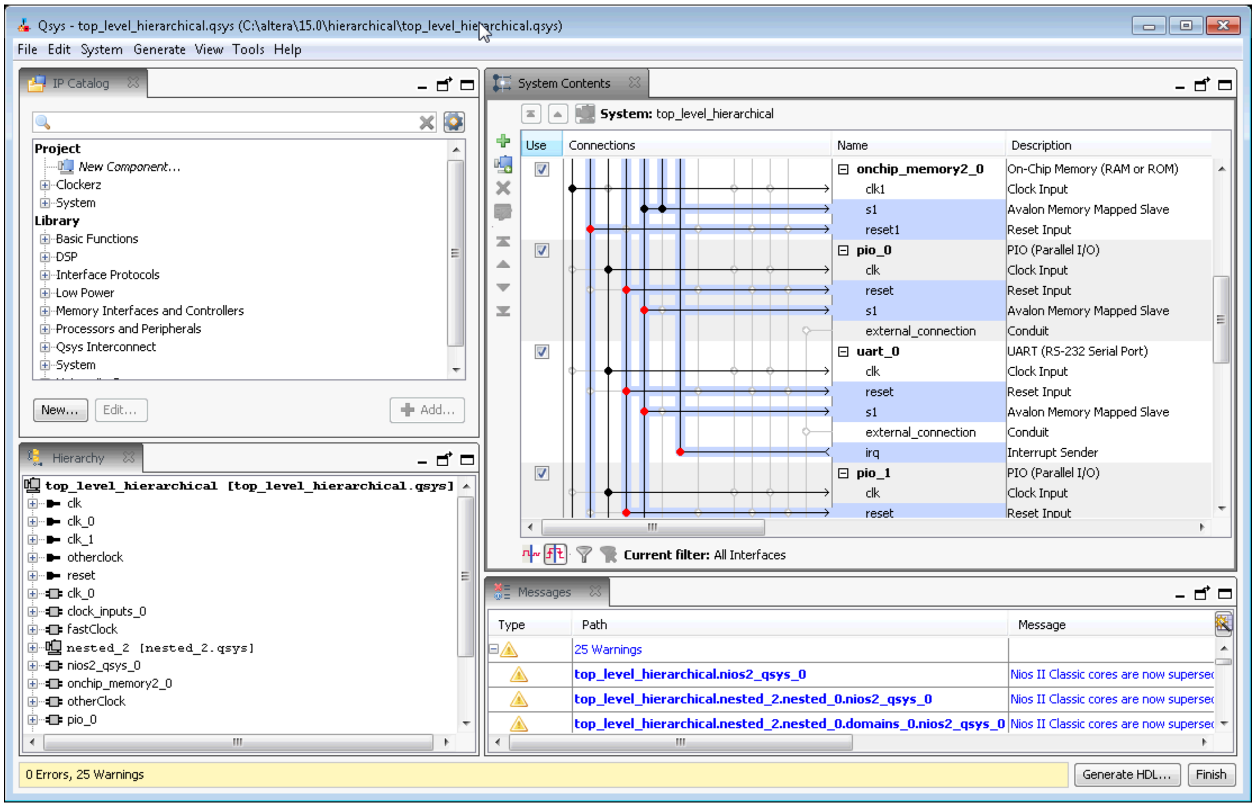
The **Messages** tab displays in the lower portion of Qsys. Double-clicking a message in the **Messages** tab changes focus to the associated element in the relevant tab to facilitate debugging. When the **Messages** tab is closed or not open in your workspace, error and warning message counts continue 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 you to explore their usefulness in Qsys system development.

Figure 4-5: View Your Qsys System



Qsys - top_level_hierarchical.qsys (C:\altera\15.0\hierarchical\top_level_hierarchical.qsys)

System: top_level_hierarchical

Use	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

Current filter: All Interfaces

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

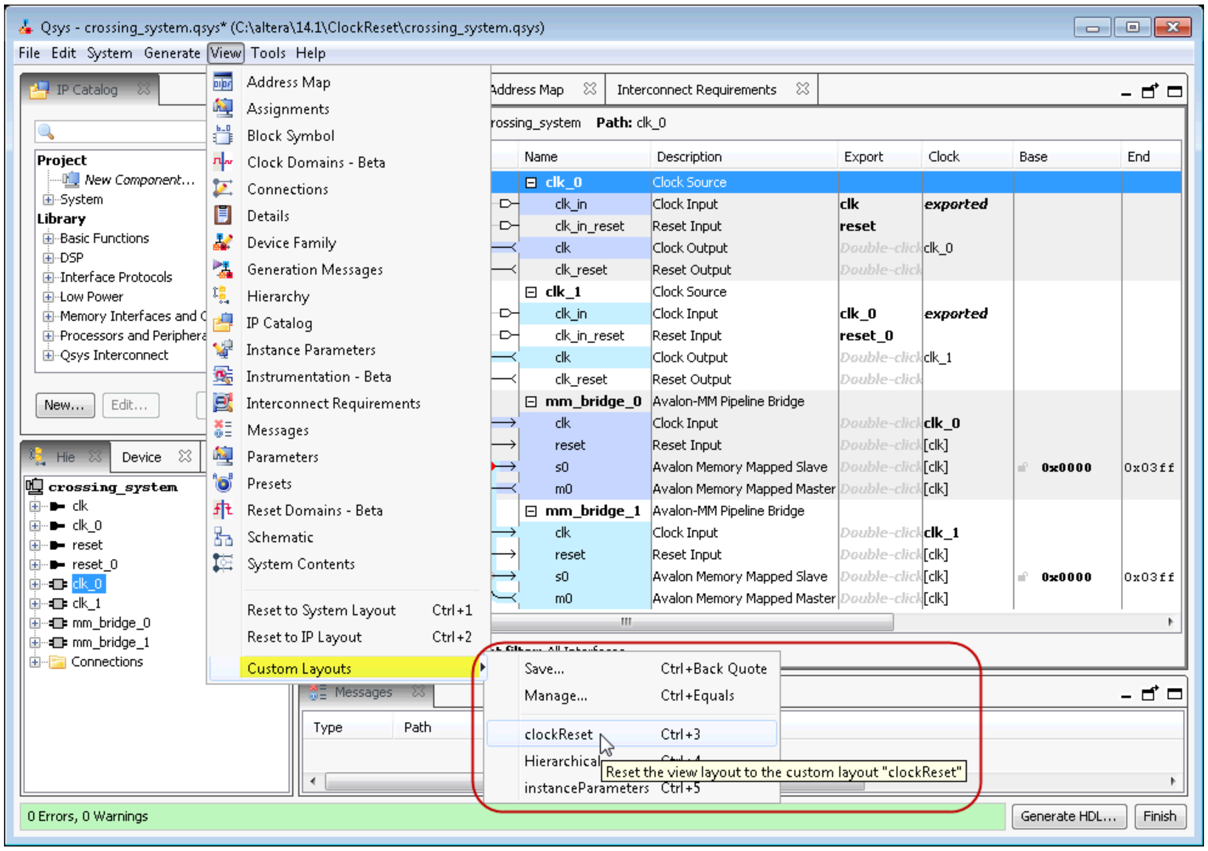
0 Errors, 25 Warnings

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. The Qsys GUI and features are well-suited for Qsys system design. Though, you can also use Qsys to define and generate single IP cores for use in your Quartus Prime 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 4-6: Save Your Qsys Window Views and Layouts



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]		

Reset the view layout to the custom layout "clockReset"

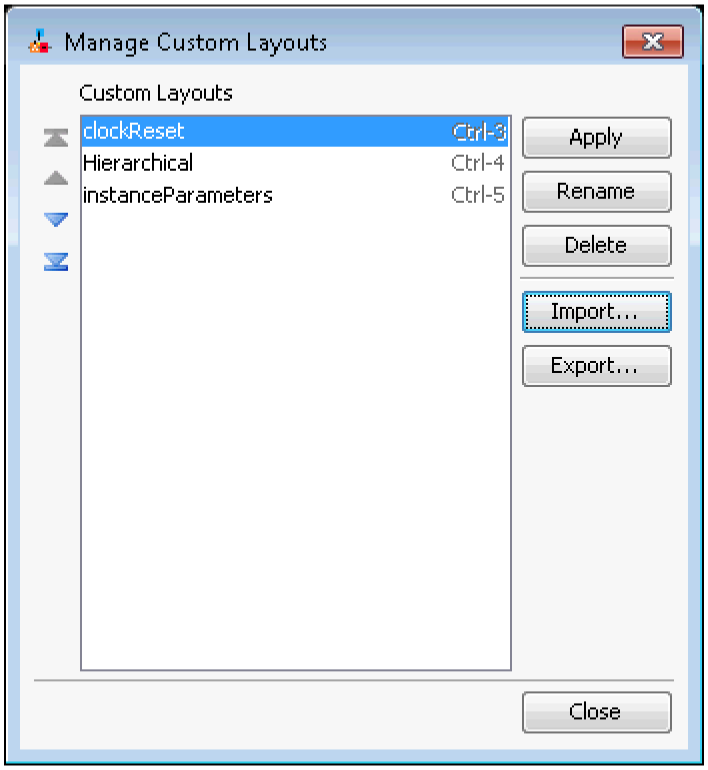
0 Errors, 0 Warnings

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. Foe example, you can import or export a layout from or to a different directory.

Figure 4-7: Manage Custom Layouts

The shortcut, **Ctrl-3**, for example, allows you to quickly change your Qsys window view with a quick keystroke.

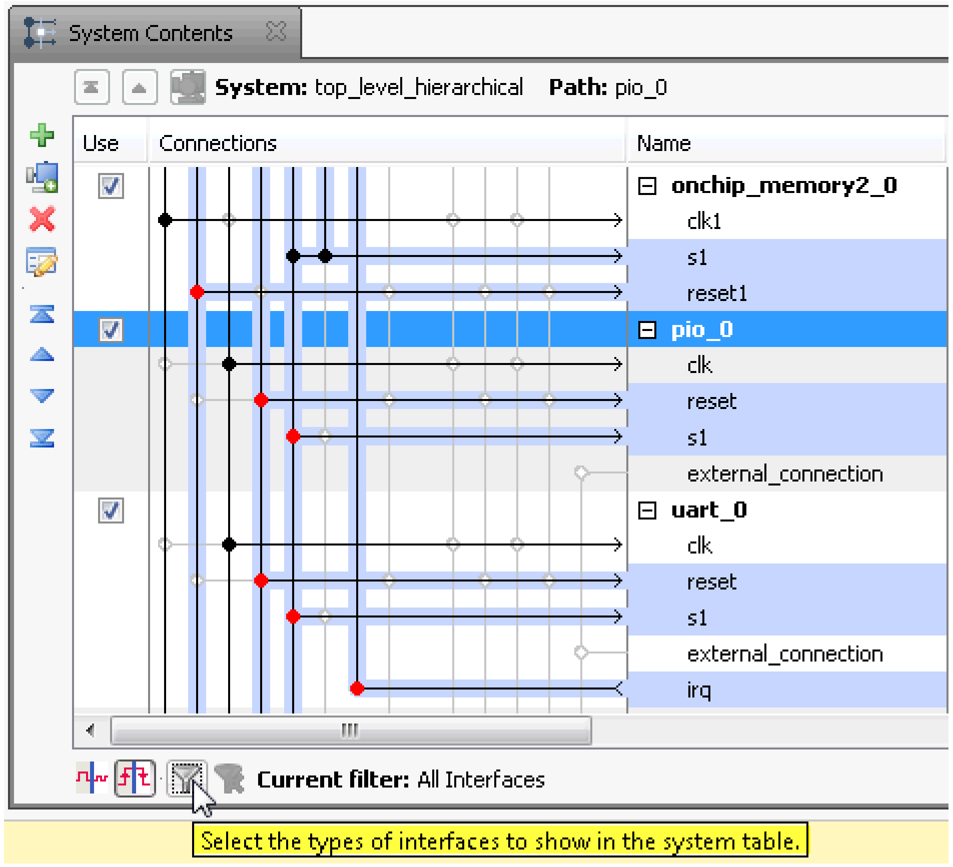


Filter the Display of the System Contents Tab

You can use the **Filters** dialog box 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 or instances connected to a particular Nios II processor. The filter tool also allows you to temporarily hide clock and reset interfaces to simplify the display.

Figure 4-8: 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.

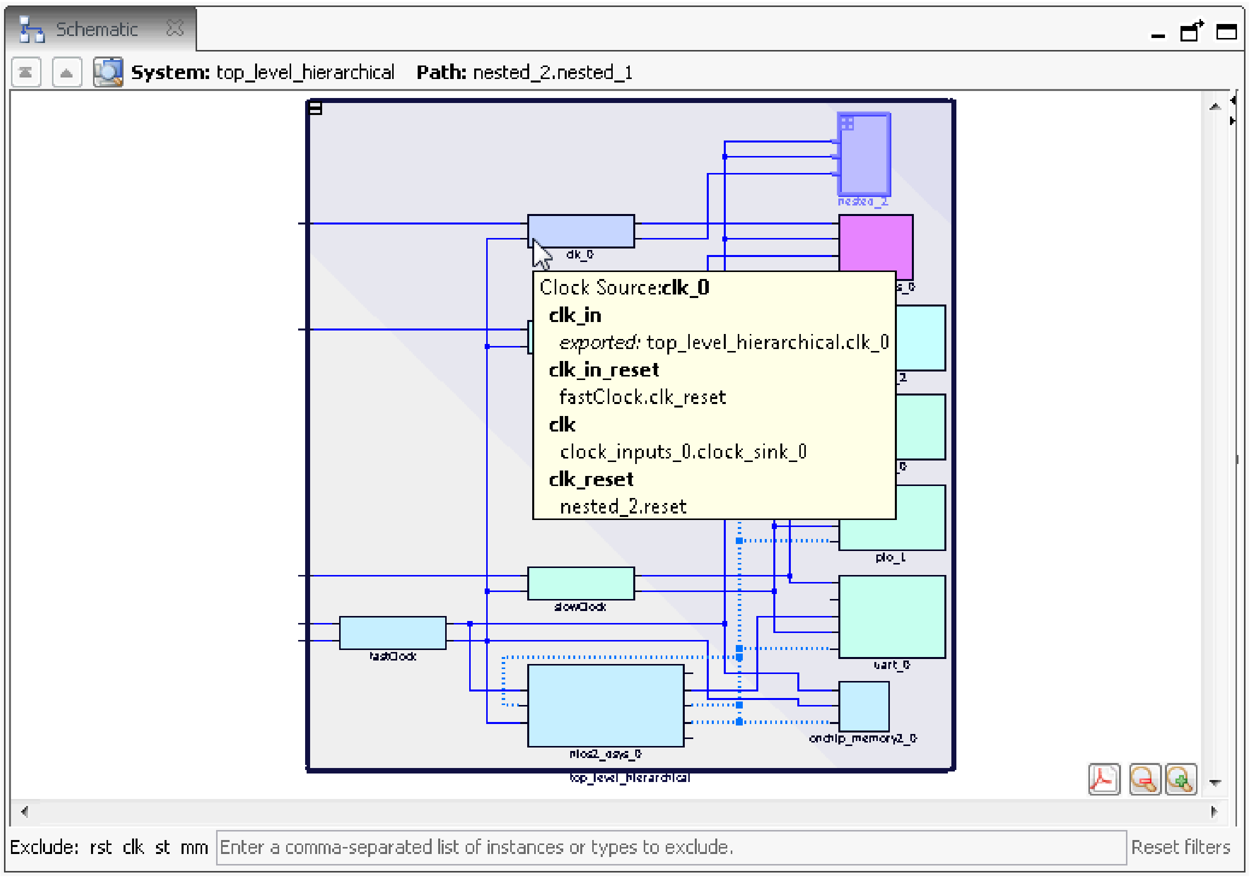
The **Block Symbol** tab appears by default in the parameter editor when you add a component to your system. When the **Block Symbol** tab is open in your workspace, it reflects changes that you make in other tabs.

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 4-9: 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. Reset filters

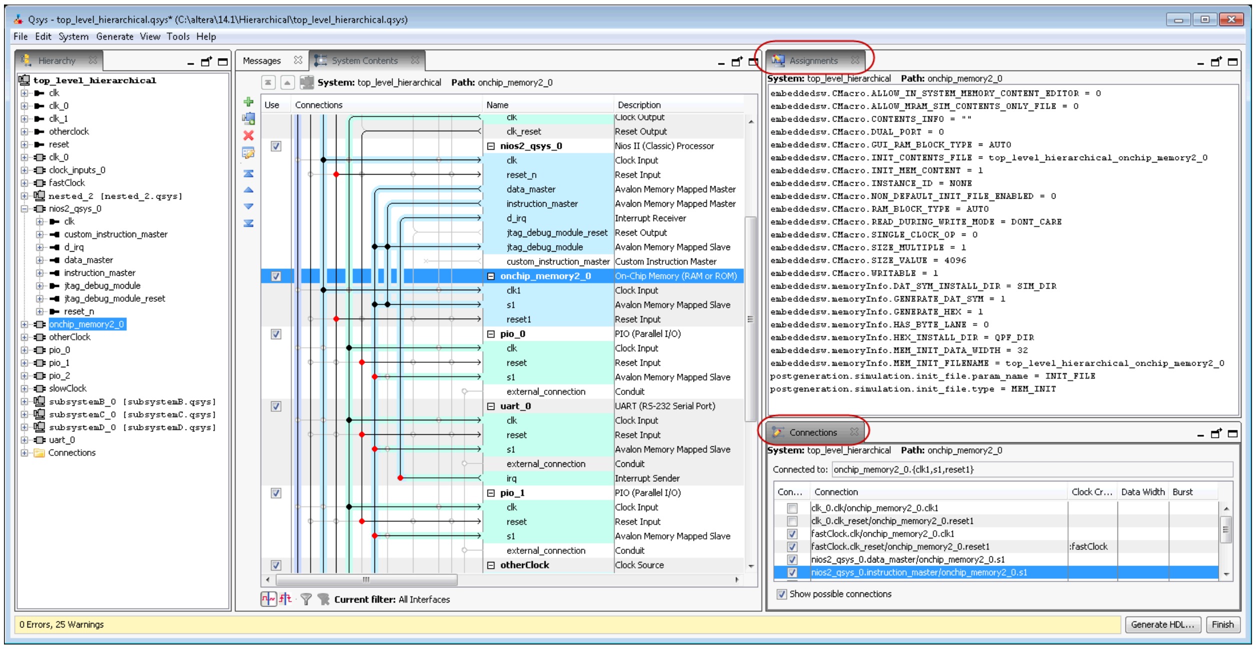
Related Information

Edit a Qsys Subsystem on page 4-36

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 4-10: 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
0 Errors, 25 Warnings
Generate HDL...
Finish

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

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.

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.

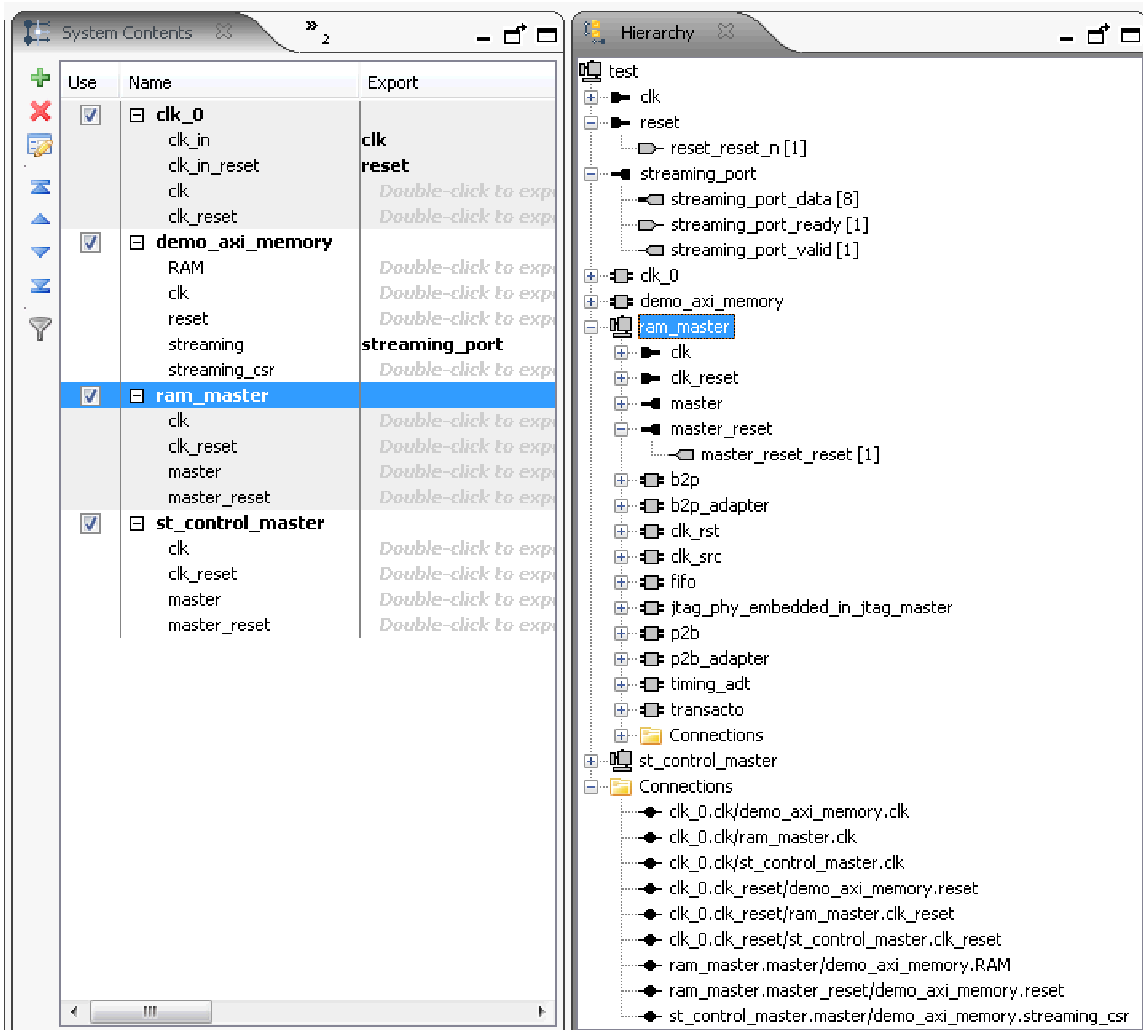
Navigating your system using the **Hierarchy** tab in conjunction with relevant tabs is useful during the debugging phase. Viewing your system with mutiple tabs open allows you to focus your debugging efforts to a single element in your system.

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 4-11: 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 4-34

Specify IP Component Parameters

The **Parameters** tab allows you to configure parameters that define an 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 opens. In the parameter editor, you can configure the parameters of the component to to align with the requirements of your design.. If you create your own IP components, use the Hardware Component Description File (**_hw.tcl**) to specify configurable parameters.

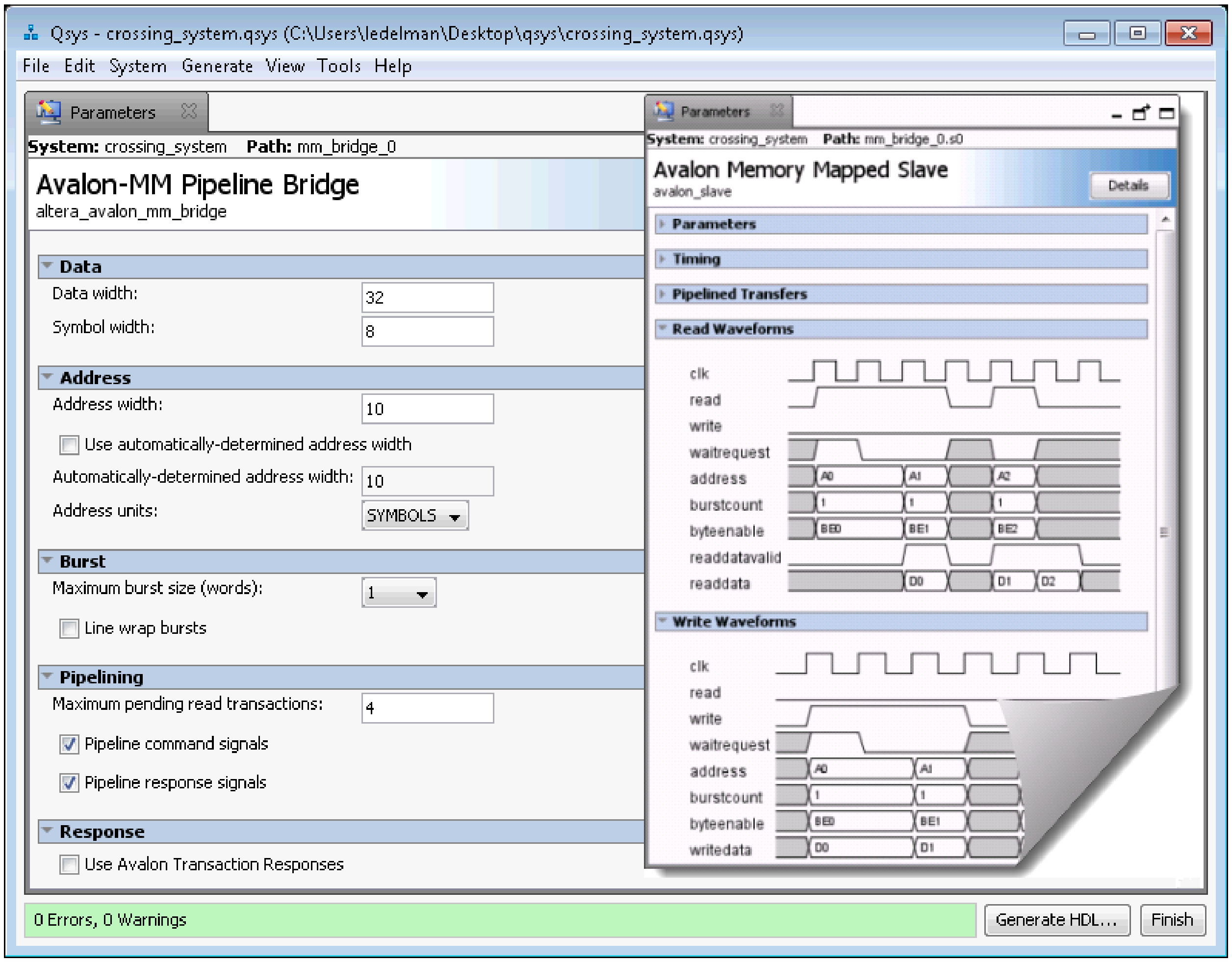
With the **Parameters** tab open, when you select an element in the **Hierarchy** tab, Qsys shows the same element in the **Parameters** tab. You can then make changes to the parameters that appear in the parameter editor, including changing the name for top-level instance that appears in the **System 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.

The **Parameters** tab also allows you to review the timing for an interfac and displays the read and write waveforms at the bottom of the **Parameters** tab.

Figure 4-12: Avalon-MM Write Master Timing Waveforms in the Parameters Tab


Qsys - crossing_system.qsys (C:\Users\ledelman\Desktop\qsys\crossing_system.qsys)
File Edit System Generate View Tools Help
Parameters
Parameters
Details
Generate HDL...
Finish

System: crossing_system **Path:** mm_bridge_0

Avalon-MM Pipeline Bridge
altera_avalon_mm_bridge

Data
Data width: 32
Symbol width: 8

Address
Address width: 10
☐ Use automatically-determined address width
Automatically-determined address width: 10
Address units: SYMBOLS

Burst
Maximum burst size (words): 1
☐ Line wrap bursts

Pipelining
Maximum pending read transactions: 4
☑ Pipeline command signals
☑ Pipeline response signals

Response
☐ Use Avalon Transaction Responses

0 Errors, 0 Warnings

System: crossing_system **Path:** mm_bridge_0.s0

Avalon Memory Mapped Slave
avalon_slave

Parameters
Timing
Pipelined Transfers
Read Waveforms

Write Waveforms

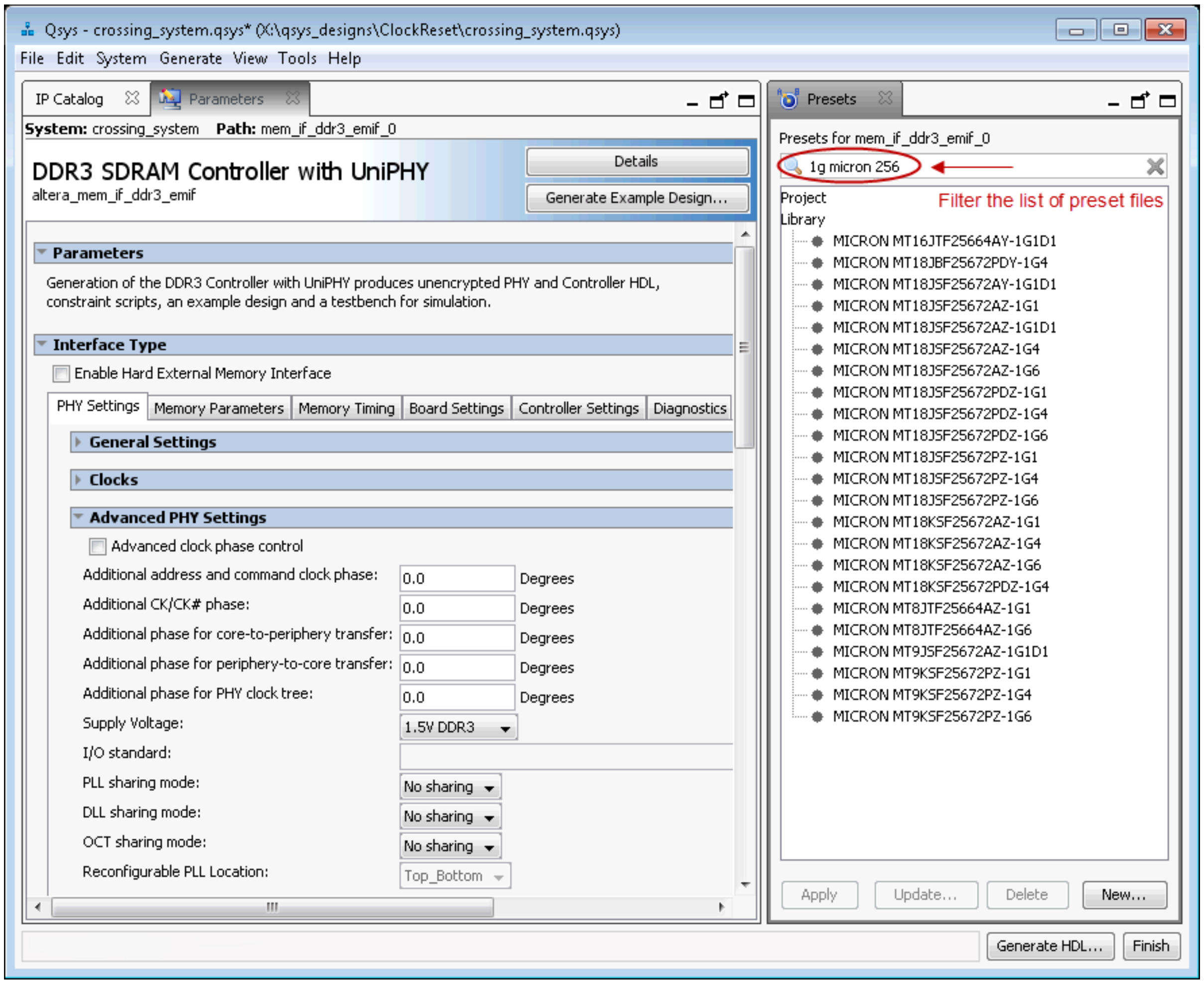
Configure Your IP Component with a Pre-Defined Set of Parameters

The **Presets** tab allows you to apply a pre-defined set of parameters to your IP component to create a unique variation. 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 files available for the component, the preset editor opens in the parameter editor. The name of each preset file describes a particular protocol.

1. In your Qsys system, select an element in the **Hierarchy** tab.
2. Click **View** > **Presets**.
3. Type text in the **Presets** search box to filter the list of preset files.
   For example, if you add the **DDR3 SDRAM Controller with UniPHY** component to your system, type `1g micron 256` in the search box, The **Presets** list displays only those preset files associated with `1g micron 256`.
4. Click **Apply** to assign the selected presets to the component.
   Presets whose parameter values match the current parameter settings appear in bold.
5. In the **Presets** tab, click **New** to create a custom preset file if the available presets do not meet the requirements of your design.
   a. In the **New Preset** dialog box, specify the **Preset name** and **Preset description**.
   b. In the Ceck or uncheck the parameters you want to include in the preset file.
   c. Specify where you want to save the new 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.
   d. Click **Save**.
6. In the **Presets** tab, click **Update** to update a custom preset.

   Note: Custom presets are preset files that you create by clicking **New** in the **Presets** tab.
7. In the **Presets** tab, click **Delete** to delete a custom preset.


Qsys - crossing_system.qsys* (X:\qsys_designs\ClockReset\crossing_system.qsys)

File Edit System Generate View Tools Help

IP Catalog | Parameters

System: crossing_system Path: mem_if_ddr3_emif_0

DDR3 SDRAM Controller with UniPHY
altera_mem_if_ddr3_emif

Details | Generate Example Design...

Parameters

Generation of the DDR3 Controller with UniPHY produces unencrypted PHY and Controller HDL, constraint scripts, an example design and a testbench for simulation.

Interface Type

☐ Enable Hard External Memory Interface

PHY Settings | Memory Parameters | Memory Timing | Board Settings | Controller Settings | Diagnostics

General Settings

Clocks

Advanced PHY Settings

☐ Advanced clock phase control

Additional address and command clock phase:	0.0	Degrees
Additional CK/CK# phase:	0.0	Degrees
Additional phase for core-to-periphery transfer:	0.0	Degrees
Additional phase for periphery-to-core transfer:	0.0	Degrees
Additional phase for PHY clock tree:	0.0	Degrees
Supply Voltage:	1.5V DDR3	
I/O standard:		
PLL sharing mode:	No sharing	
DLL sharing mode:	No sharing	
OCT sharing mode:	No sharing	
Reconfigurable PLL Location:	Top_Bottom	

Presets

Presets for mem_if_ddr3_emif_0

1g micron 256

Filter the list of preset files

Project
Library
- MICRON MT16JTF25664AY-1G1D1
- MICRON MT18JBF25672PDY-1G4
- MICRON MT18JSF25672AY-1G1D1
- MICRON MT18JSF25672AZ-1G1
- MICRON MT18JSF25672AZ-1G1D1
- MICRON MT18JSF25672AZ-1G4
- MICRON MT18JSF25672AZ-1G6
- MICRON MT18JSF25672PDZ-1G1
- MICRON MT18JSF25672PDZ-1G4
- MICRON MT18JSF25672PDZ-1G6
- MICRON MT18JSF25672PZ-1G1
- MICRON MT18JSF25672PZ-1G4
- MICRON MT18JSF25672PZ-1G6
- MICRON MT18KSF25672AZ-1G1
- MICRON MT18KSF25672AZ-1G4
- MICRON MT18KSF25672AZ-1G6
- MICRON MT18KSF25672PDZ-1G4
- MICRON MT8JTF25664AZ-1G1
- MICRON MT8JTF25664AZ-1G6
- MICRON MT9JSF25672AZ-1G1D1
- MICRON MT9KSF25672PZ-1G1
- MICRON MT9KSF25672PZ-1G4
- MICRON MT9KSF25672PZ-1G6

Apply | Update... | Delete | New...

Generate HDL... | Finish

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.

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 Prime 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 4-3: 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 8-1

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.

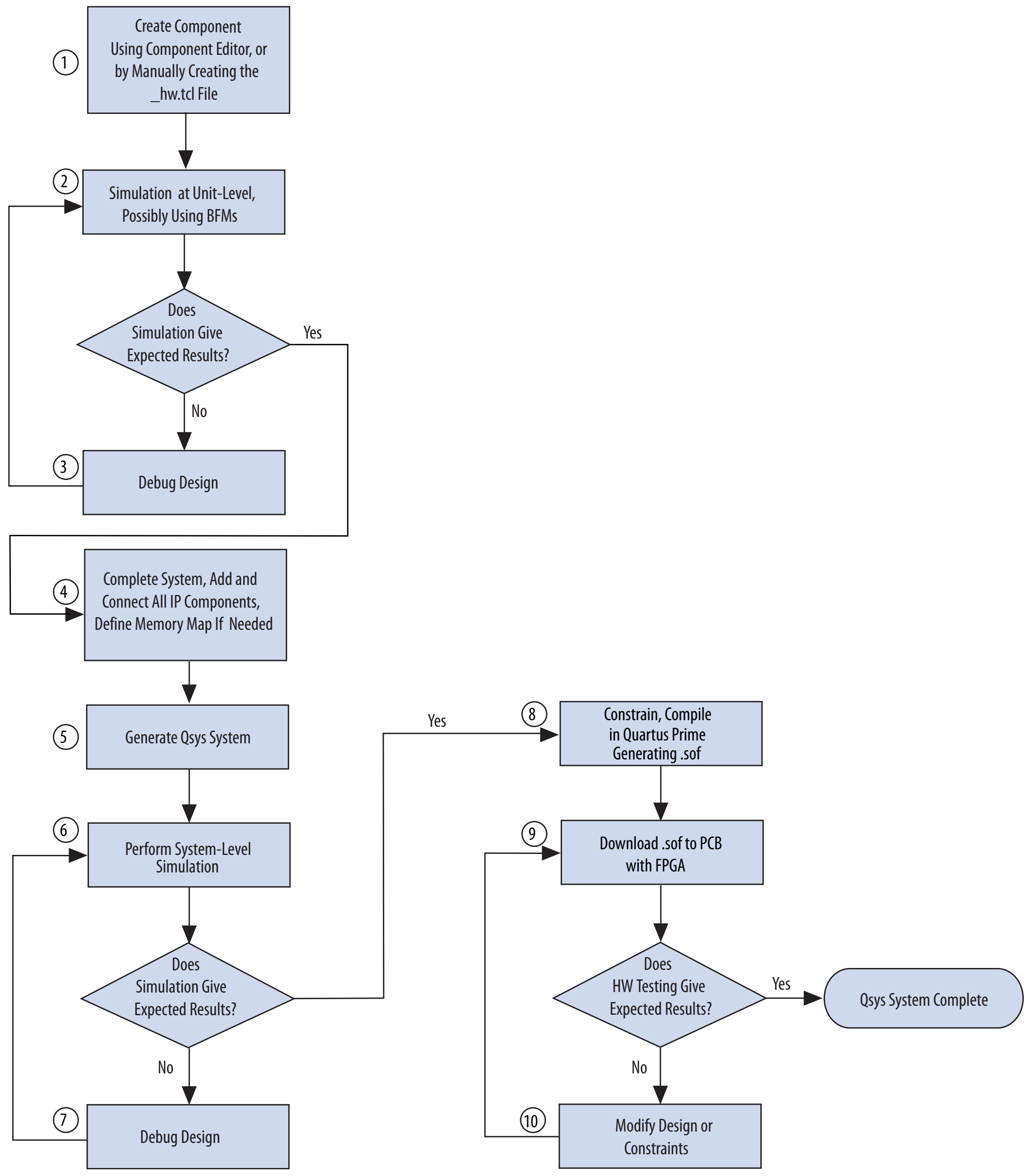
Example

```
set_module_property COMPOSITION_CALLBACK "my_composition_callback"
```

Create a Custom IP Component (_hw.tcl)

Figure 4-13: 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 Prime 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?
No
7
Debug Design
Yes
8
Constrain, Compile in Quartus Prime Generating .sof
9
Download .sof to PCB with 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 Prime software projects, refer to *Introduction to Altera IP Cores*.

Related Information

Creating Qsys Components on page 5-1

Introduction to Altera IP Cores

Managing Quartus Prime Projects on page 2-1

Upgrade Outdated IP Components in Qsys

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 upgradess 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 upgraded 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, or the directory of the **.qsys** file, you must add the location of these components 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. You do not need to run 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

Introduction to the Qsys IP Catalog on page 4-3

Introduction to Altera IP Cores

Troubleshooting IP or Qsys System Upgrade

The **Upgrade IP Components** dialog box reports the version and status of each IP core and Qsys system following upgrade or migration. If any upgrade or migration fails, the **Upgrade IP Components** dialog box provides information to help you resolve any errors.

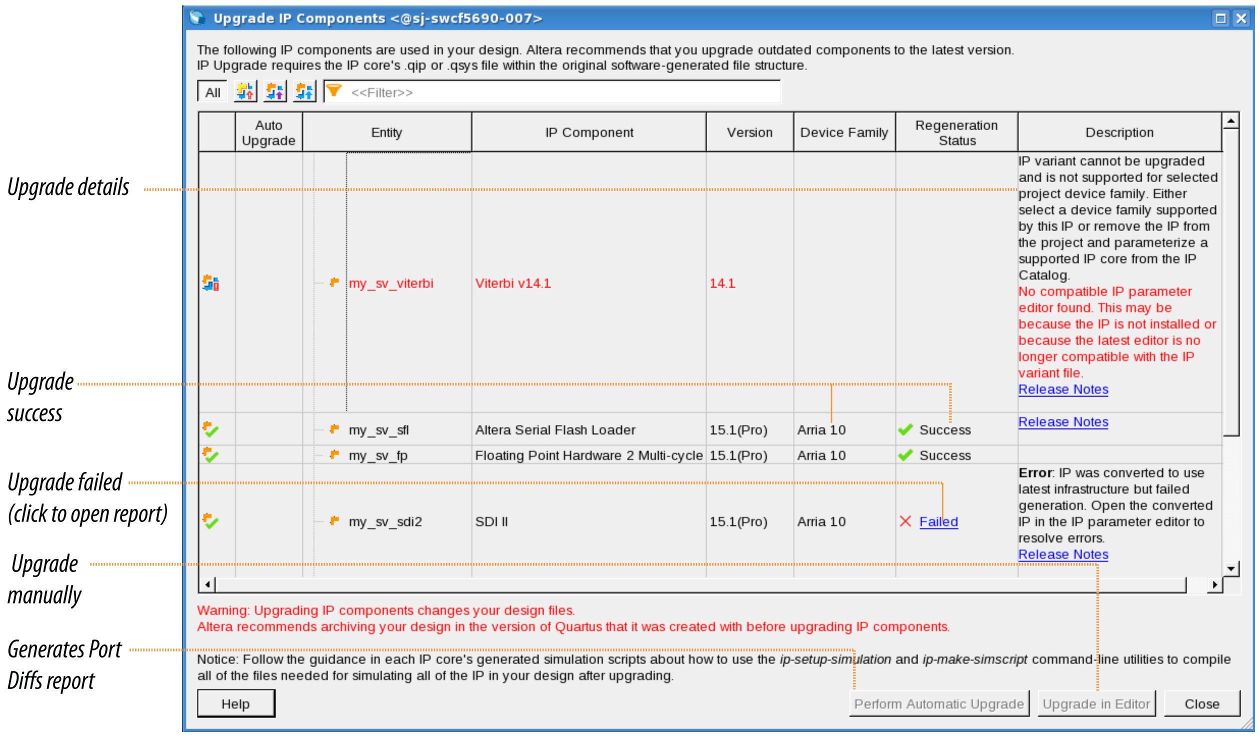
Note: Make sure that your IP variation names or paths do not include spaces. Spaces can be problematic for IP generation.

During automatic or manual upgrade, the Messages window dynamically displays upgrade information for each IP core or Qsys system. You can use the following information to help you resolve any upgrade errors following upgrade or migration.

Table 4-4: IP Upgrade Error Information

Upgrade IP Components Field	Description
Regneration Status	Displays the "Success" or "Failed" status of each upgrade or migration. Click the status of any failed upgrade to open a detailed **IP Upgrade Report**.
Version	Dynamically updates to the new version number when upgrade is successful. The text is red when upgrade is required.
Device Family	Dynamically updates to the new device family when migration is successful. The text is red when upgrade is required.
Description	Summarizes IP release information and displays actionable, corrective action for resolving upgrade or migration failures. Follow these instructions to resolve upgrade failures. Click the **Release Notes** link for the latest known issues about the Altera IP core.
Perform Automatic Upgrade	Runs automatic upgrade on all IP cores that support auto upgrade. Also, automatically generates a *<Project Directory>*/**ip_upgrade_port_diff_report** report for IP cores or Qsys systems that fail upgrade. Review these reports to determine any port differences between the current and previous IP core version.

Figure 4-14: Resolving Upgrade Errors


Upgrade details
Upgrade success
Upgrade failed (click to open report)
Upgrade manually
Generates Port Diffs report
Upgrade IP Components <@sj-swcf5690-007>
All
<<Filter>>
Help
Perform Automatic Upgrade
Upgrade in Editor
Close

The following IP components are used in your design. Altera recommends that you upgrade outdated components to the latest version. IP Upgrade requires the IP core's .qip or .qsys file within the original software-generated file structure.

Auto Upgrade	Entity	IP Component	Version	Device Family	Regeneration Status	Description
	my_sv_viterbi	Viterbi v14.1	14.1			IP variant cannot be upgraded and is not supported for selected project device family. Either select a device family supported by this IP or remove the IP from the project and parameterize a supported IP core from the IP Catalog. No compatible IP parameter editor found. This may be because the IP is not installed or because the latest editor is no longer compatible with the IP variant file. Release Notes
	my_sv_sfl	Altera Serial Flash Loader	15.1(Pro)	Arria 10	✓ Success	Release Notes
	my_sv_fp	Floating Point Hardware 2 Multi-cycle	15.1(Pro)	Arria 10	✓ Success	
	my_sv_sdi2	SDI II	15.1(Pro)	Arria 10	✗ Failed	**Error**: IP was converted to use latest infrastructure but failed generation. Open the converted IP in the IP parameter editor to resolve errors. Release Notes

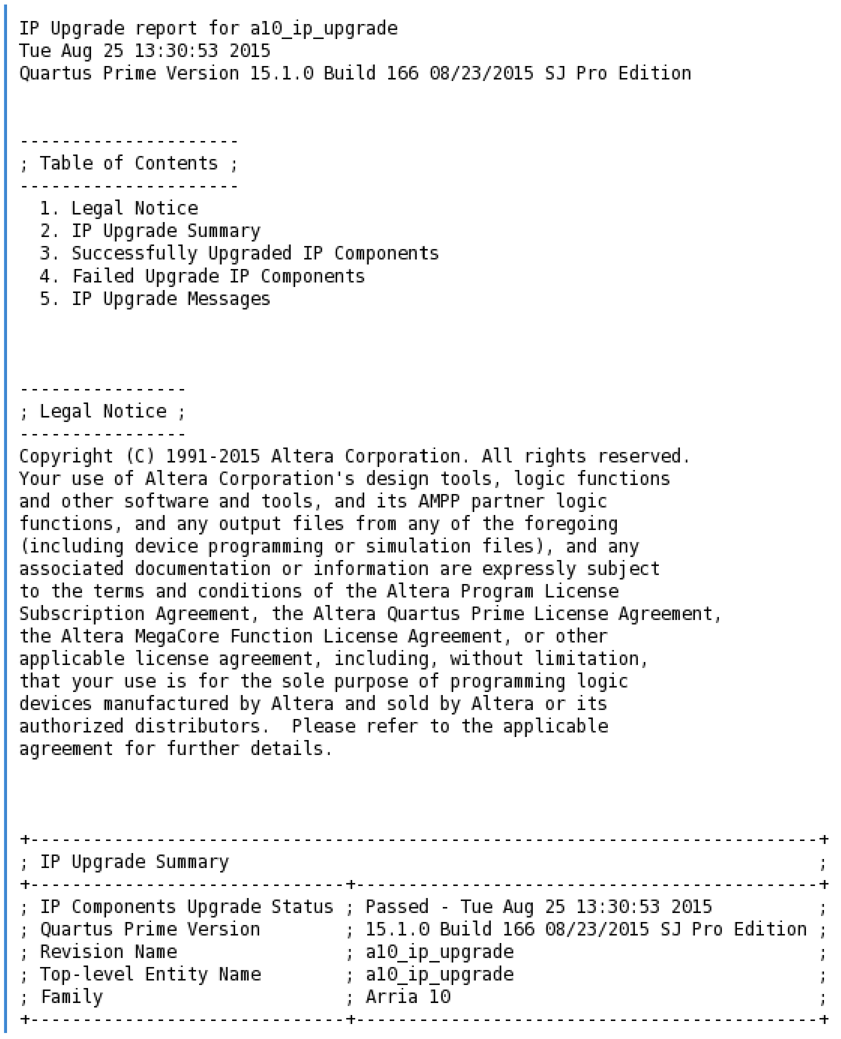
Warning: Upgrading IP components changes your design files.
Altera recommends archiving your design in the version of Quartus that it was created with before upgrading IP components.

Notice: Follow the guidance in each IP core's generated simulation scripts about how to use the *ip-setup-simulation* and *ip-make-simscript* command-line utilities to compile all of the files needed for simulating all of the IP in your design after upgrading.

Use the following techniques to resolve errors if your Altera IP core or Qsys system "Failed" to upgrade versions or migrate to another device. Review and implement the instructions in the **Description** field, including one or more of the following:

1. If the IP variant is not supported in the current version of the software, right-click the component and click **Remove IP Component from Project**. Replace this IP core or Qsys system with one supported in the current version of the software.
2. If the IP variant is not supported by the current target device, select a supported device family for the project, or replace the IP variant with a suitable replacement that supports your target device.
3. If an upgrade or migration fails, click **Failed** in the **Regeneration Status** field to display and review details of the **IP Upgrade Report**. Click the **Release Notes** link for the latest known issues about the Altera IP core. Use this information to determine the nature of the upgrade or migration failure and make corrections before upgrade.

Figure 4-15: IP Upgrade Report


```
IP Upgrade report for a10_ip_upgrade
Tue Aug 25 13:30:53 2015
Quartus Prime Version 15.1.0 Build 166 08/23/2015 SJ Pro Edition


---------------------
; Table of Contents ;
---------------------
  1. Legal Notice
  2. IP Upgrade Summary
  3. Successfully Upgraded IP Components
  4. Failed Upgrade IP Components
  5. IP Upgrade Messages



----------------
; Legal Notice ;
----------------
Copyright (C) 1991-2015 Altera Corporation. All rights reserved.
Your use of Altera Corporation's design tools, logic functions
and other software and tools, and its AMPP partner logic
functions, and any output files from any of the foregoing
(including device programming or simulation files), and any
associated documentation or information are expressly subject
to the terms and conditions of the Altera Program License
Subscription Agreement, the Altera Quartus Prime License Agreement,
the Altera MegaCore Function License Agreement, or other
applicable license agreement, including, without limitation,
that your use is for the sole purpose of programming logic
devices manufactured by Altera and sold by Altera or its
authorized distributors.  Please refer to the applicable
agreement for further details.



+----------------------------------------------------------------------------+
; IP Upgrade Summary                                                         ;
+-------------------------------+--------------------------------------------+
; IP Components Upgrade Status ; Passed - Tue Aug 25 13:30:53 2015          ;
; Quartus Prime Version         ; 15.1.0 Build 166 08/23/2015 SJ Pro Edition ;
; Revision Name                 ; a10_ip_upgrade                             ;
; Top-level Entity Name         ; a10_ip_upgrade                             ;
; Family                        ; Arria 10                                   ;
+-------------------------------+--------------------------------------------+
```

4. Run **Perform Automatic Upgrade** to automatically generate an **IP Ports Diff** report for each IP core or Qsys system that fails upgrade. Review the reports to determine any port differences between the

current and previous IP core version. Then, click **Upgrade in Editor** to make specific port changes and regenerate your IP core or Qsys system.

5. If your IP core or Qsys system does not support **Perform Automatic Upgrade**, click **Upgrade in Editor** to resolve errors and regenerate the component in the parameter editor.

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 4-17

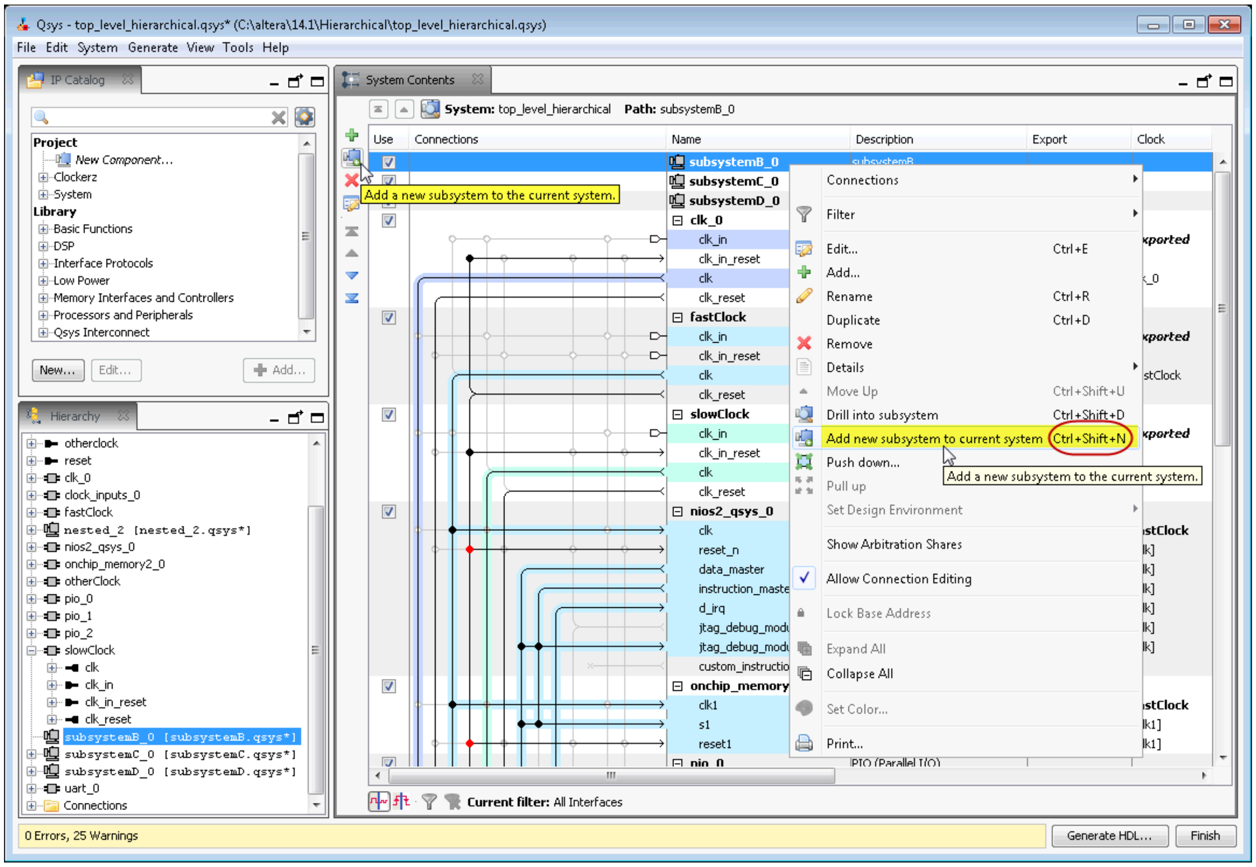
Add a Subsystem to Your Qsys Design

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 4-16: 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...
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.

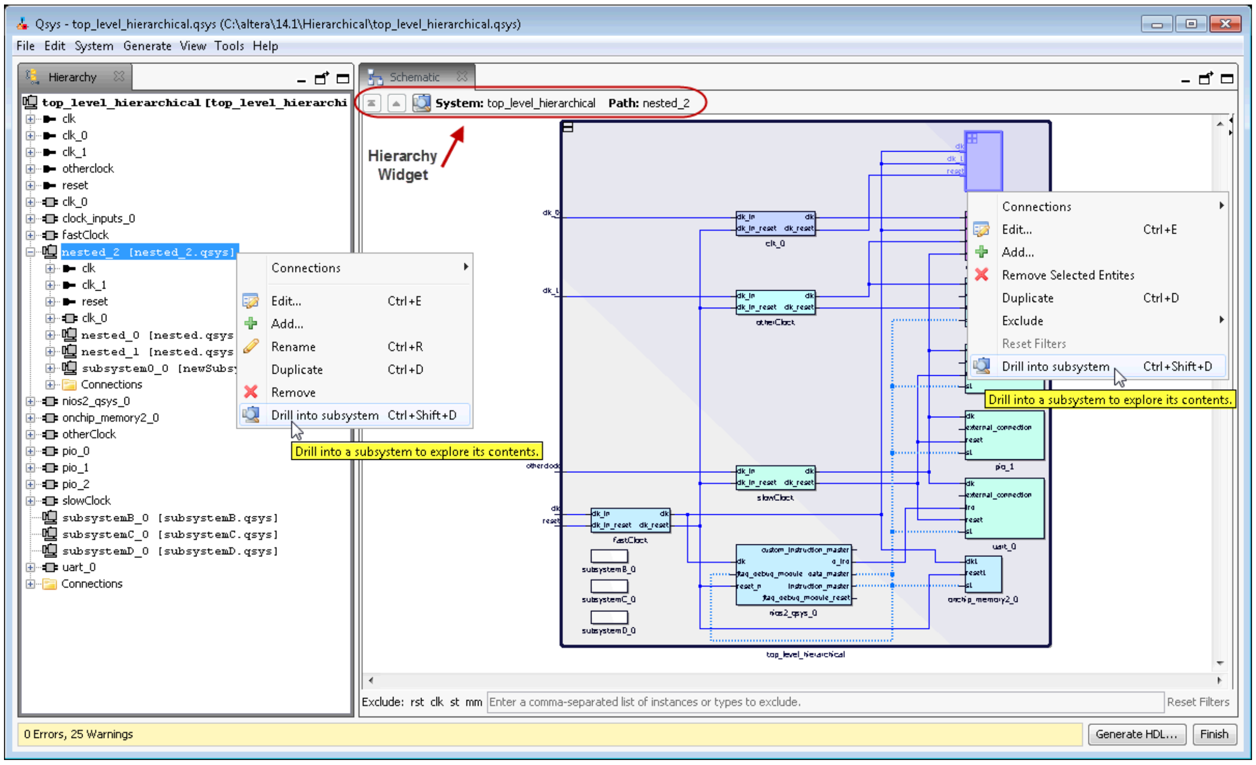
Views that manage system-level editing, for example, the **System Contents** and **Schematic** tabs, contain the hierarchy widget, which allows you to efficiently navigate your subsystems. The hierarchy widget also displays the name of the current selection, and its path in the context of the system 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.

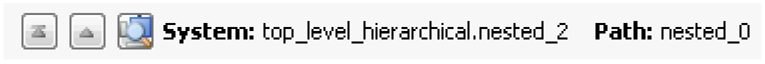
Figure 4-17: Drill into a Qsys System to Explore its Contents


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
nested_2 [nested_2.qsys]
Connections
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
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.


System: top_level_hierarchical.nested_2 **Path:** nested_0

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 4-16

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 **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.

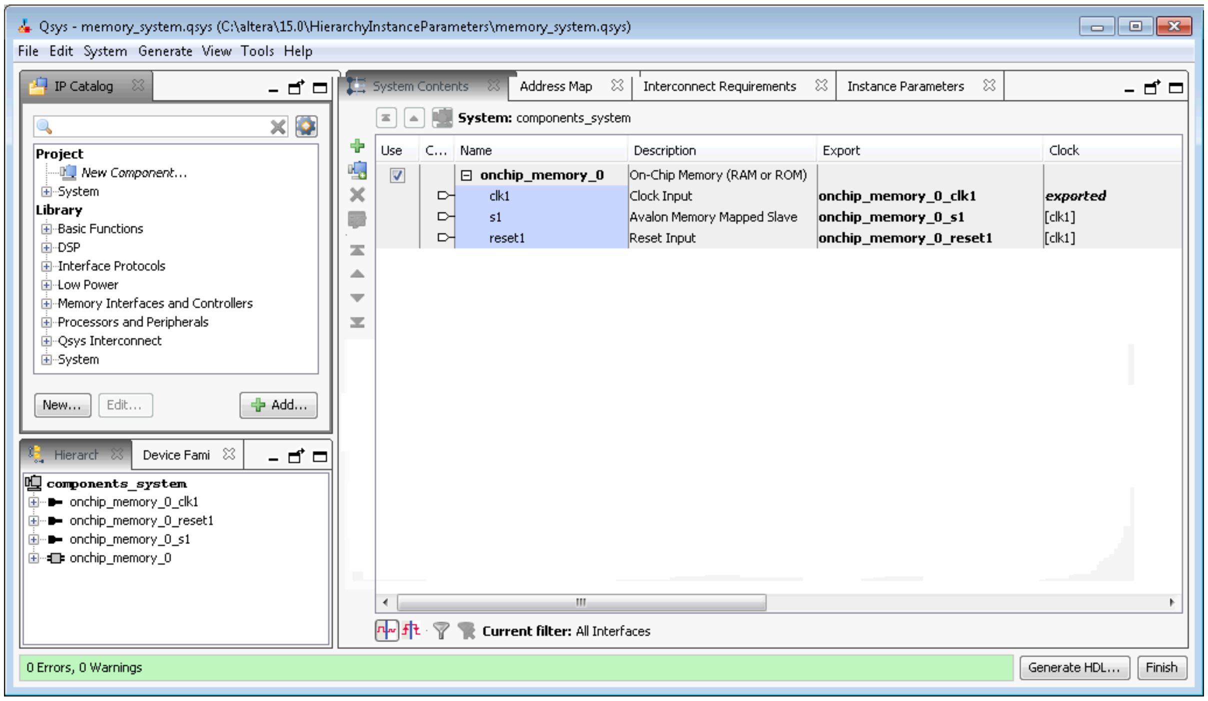
Follow the steps below to create a system that contains an on-chip memory IP component with instance parameters, and the instantiating higher-level Qsys system. With your completed system, you can vary the values of the instance parameters to review their effect within the On-Chip Memory component.

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 4-18: 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]

Hierarchy:
- components_system
  - onchip_memory_0_clk1
  - onchip_memory_0_reset1
  - onchip_memory_0_s1
  - onchip_memory_0

Current filter: All Interfaces

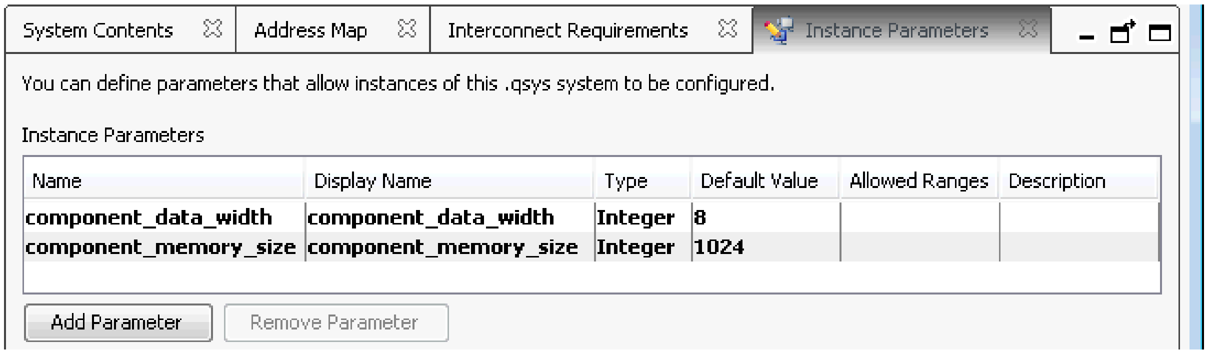
0 Errors, 0 Warnings

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 4-19: Qsys Instance Parameters Tab



You can define parameters that allow instances of this .qsys system to be configured.

Instance Parameters

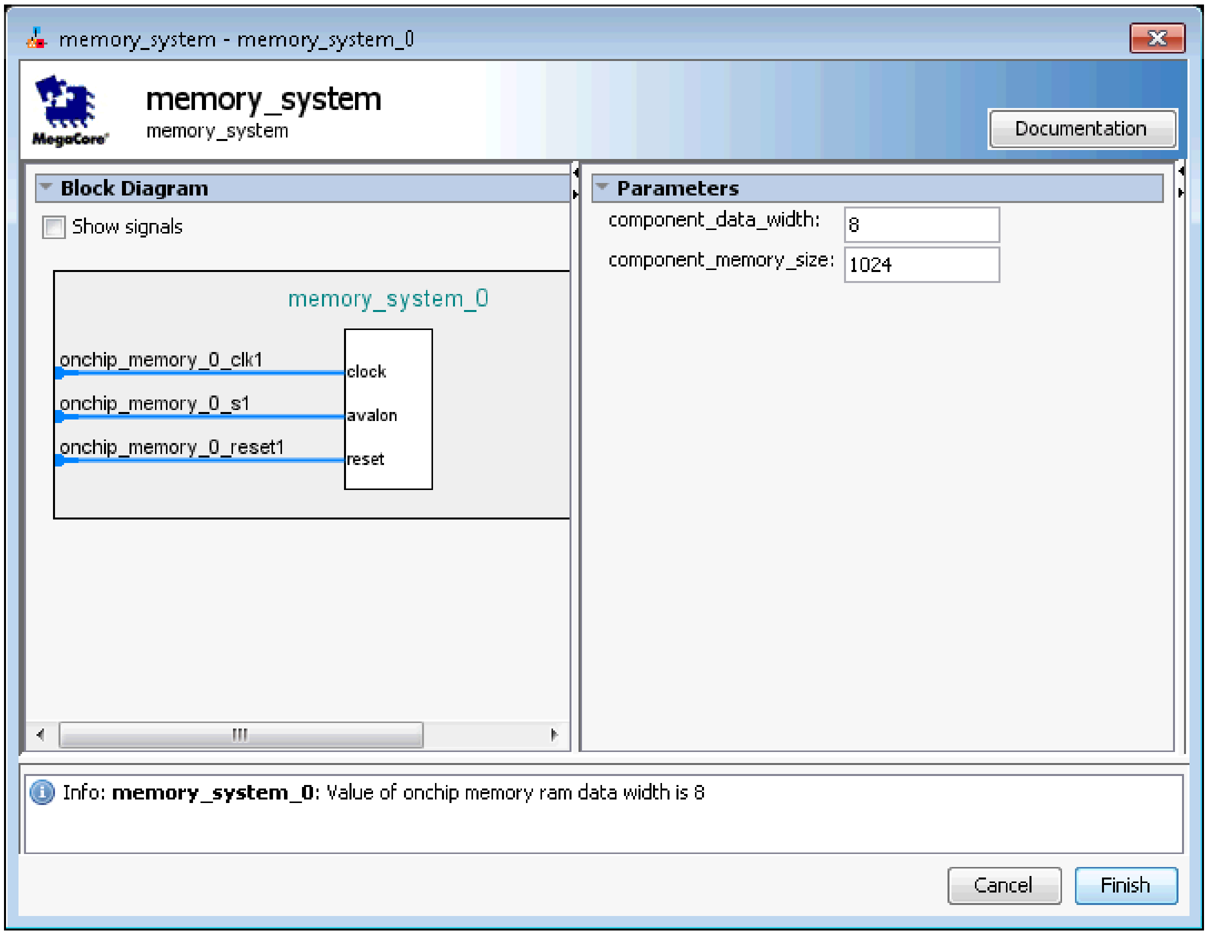
Name	Display Name	Type	Default Value	Allowed Ranges	Description
component_data_width	component_data_width	Integer	8		
component_memory_size	component_memory_size	Integer	1024		

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 4-20: 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

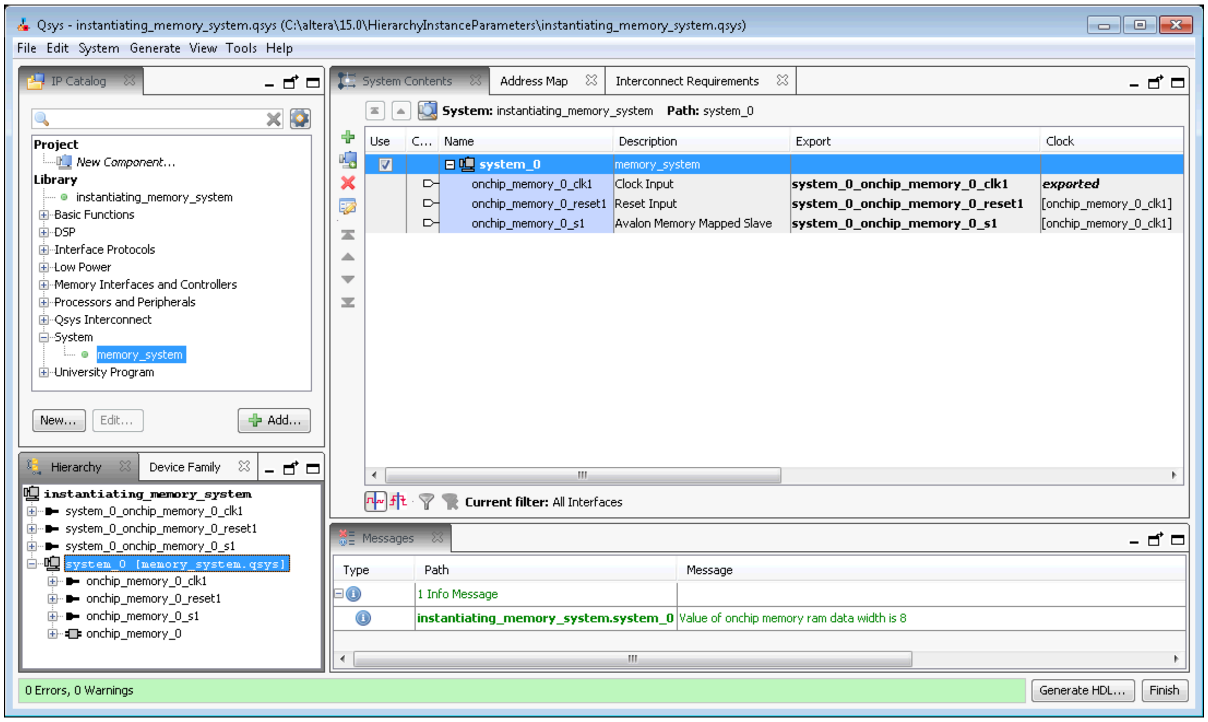
10. Click **File** > **Save**.

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 4-21: 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
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 Contents
Address Map
Interconnect Requirements
System: instantiating_memory_system Path: system_0
Current filter: All Interfaces
Messages
0 Errors, 0 Warnings
Generate HDL...
Finish

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]

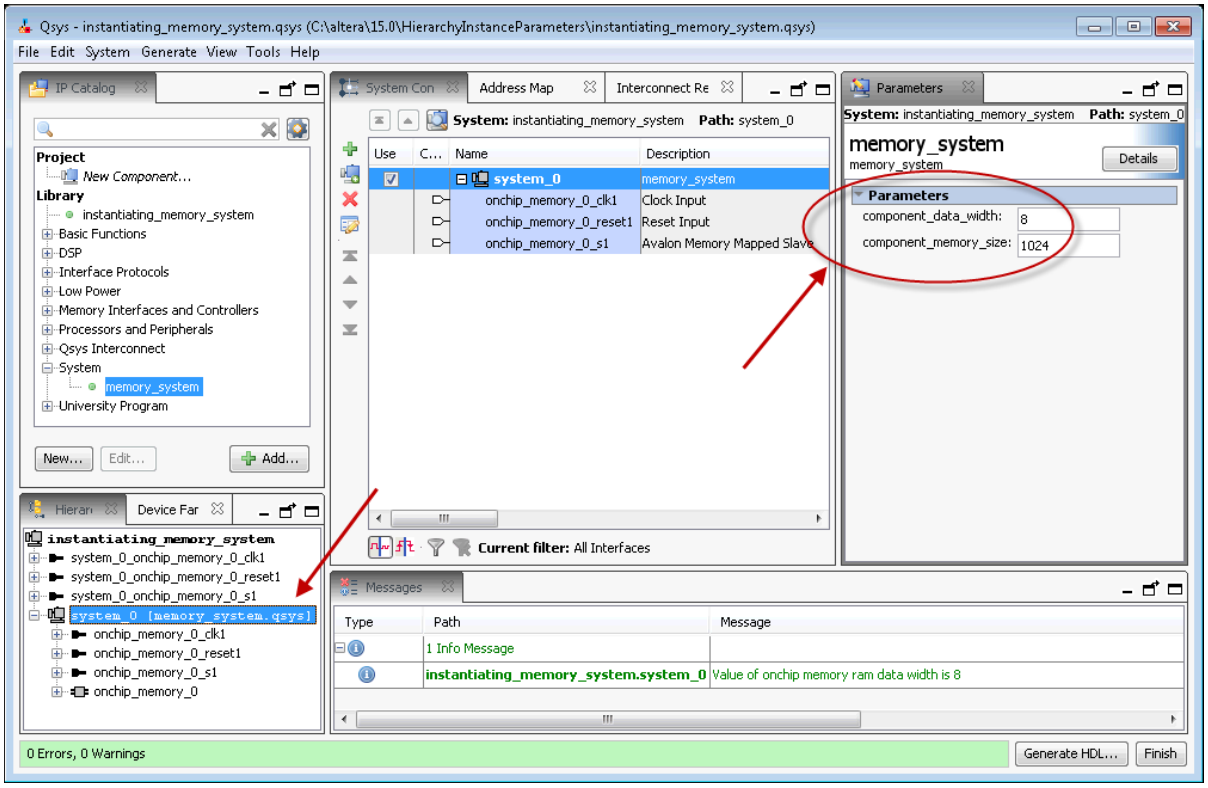
Type	Path	Message
	1 Info Message	
	instantiating_memory_system.system_0	Value of onchip memory ram data width is 8

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.

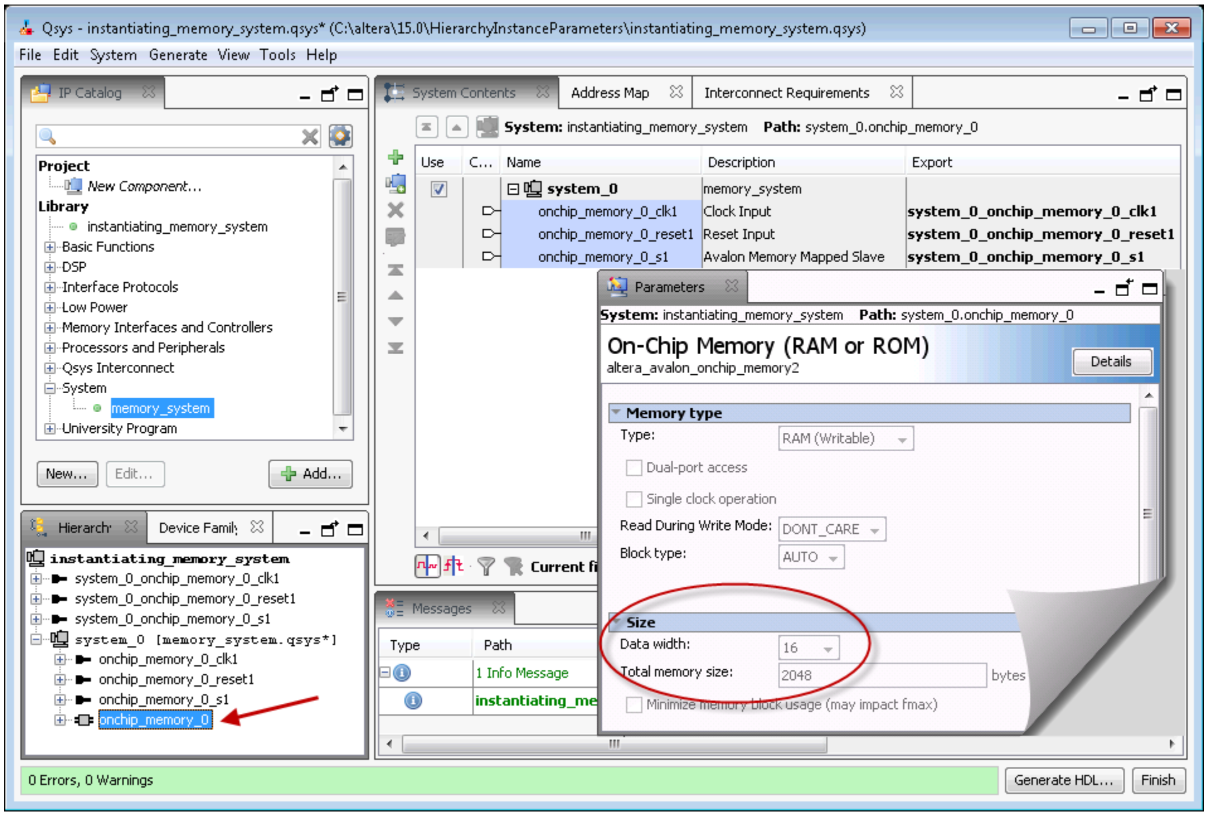
1. In the **instantiating_memory_system.qsys** system, in the **Hierarchy** tab, click and expand **system_0 (memory_system.qsys)**.
2. Click **View** > **Parameters**.
   The instance paramters for the **memory_system.qsys** display in the parameter editor.

Figure 4-22: 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 4-23: 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
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 Contents
Address Map
Interconnect Requirements
Current filter
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

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.

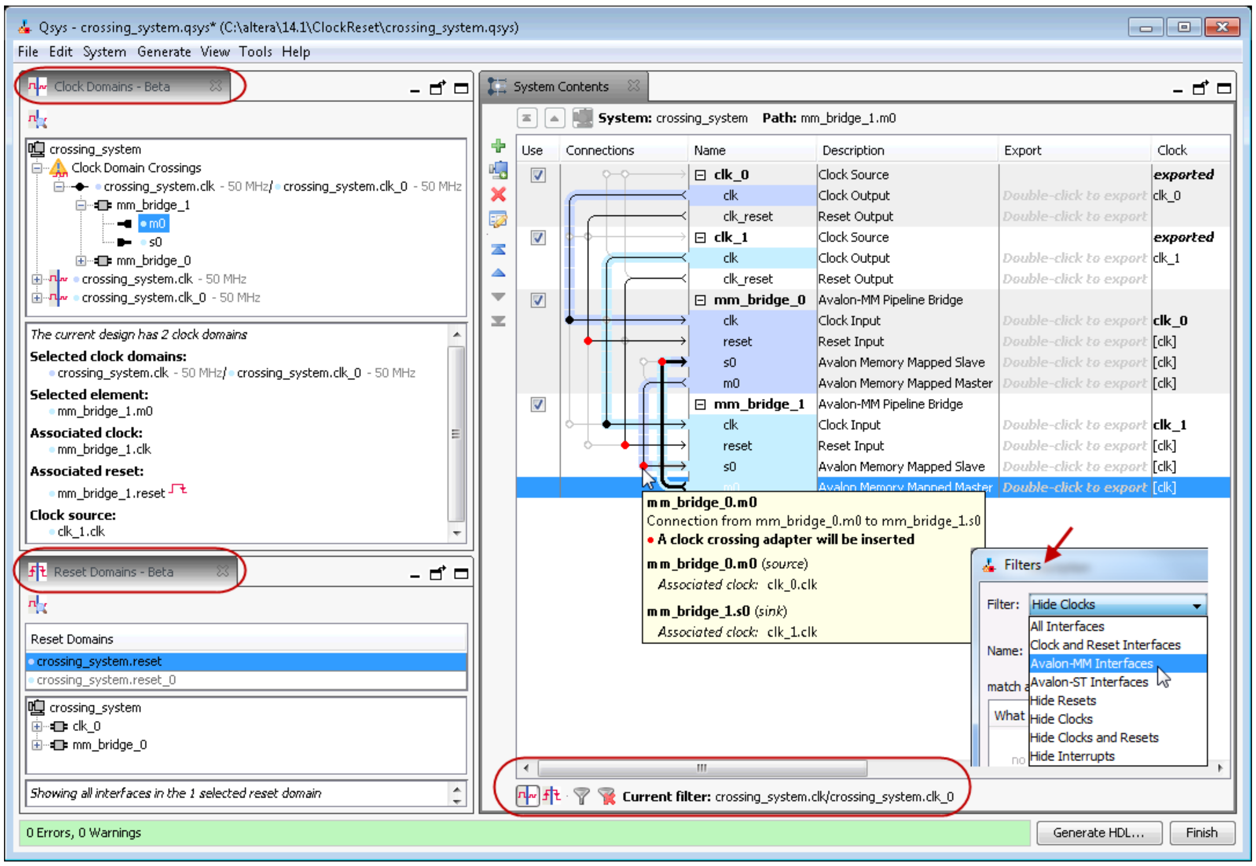
When you select an element in the **Clock Domains** tab, the corresponding selection appears in the **System Contents** tab. You can select single or multiple interface(s) and module(s). Mouse over tooltips in the **System Contents** tab to provide detailed information for all elements and connections. 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 4-24: Qsys Clock and Reset Domains



Qsys - crossing_system.qsys* (C:\altera\14.1\ClockReset\crossing_system.qsys)

Clock Domains - Beta

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 - Beta

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 Contents

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

Filters
Filter: Hide Clocks
All Interfaces
Clock and Reset Interfaces
Avalon-MM Interfaces
Avalon-ST Interfaces
Hide Resets
Hide Clocks
Hide Clocks and Resets
Hide Interrupts

Current filter: crossing_system.clk/crossing_system.clk_0

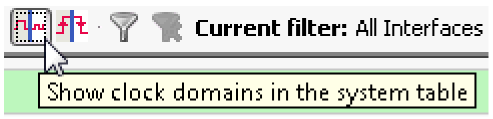
0 Errors, 0 Warnings

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.

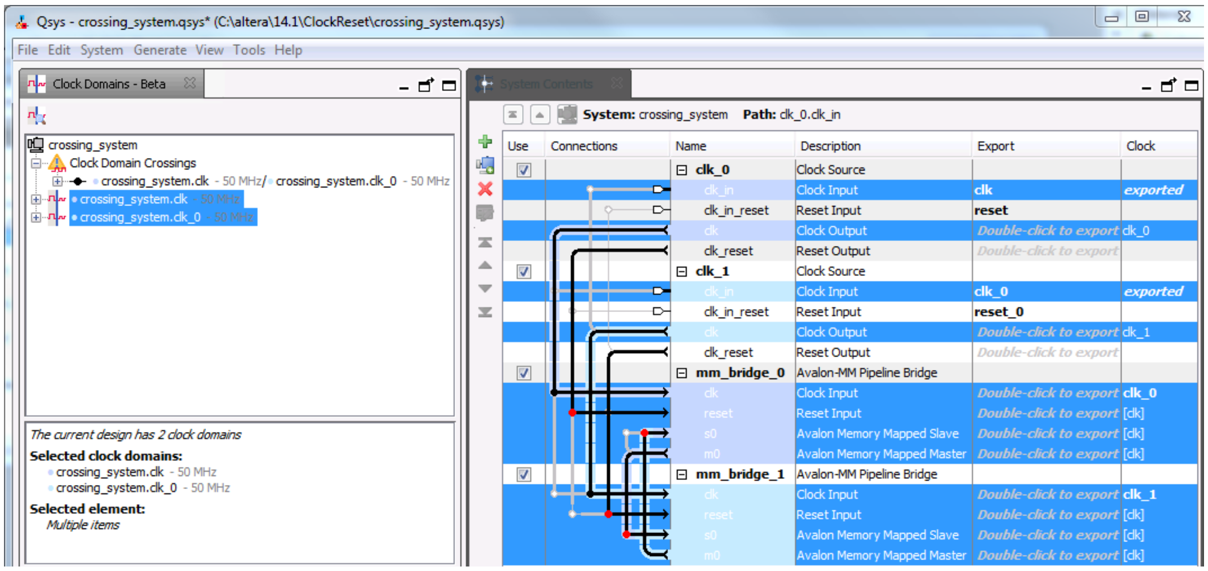
1. To view clock domain interfaces and their connections in your Qsys system, click **View** > **Clock Domains** to open the Clock Domains tab.
2. To enables and disable highlighting of the clock domains in the **System Contents** tab, click the clock control tool at the bottom of the **System Contents** tab.

Figure 4-25: 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 4-26: Clock Domains


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
- 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

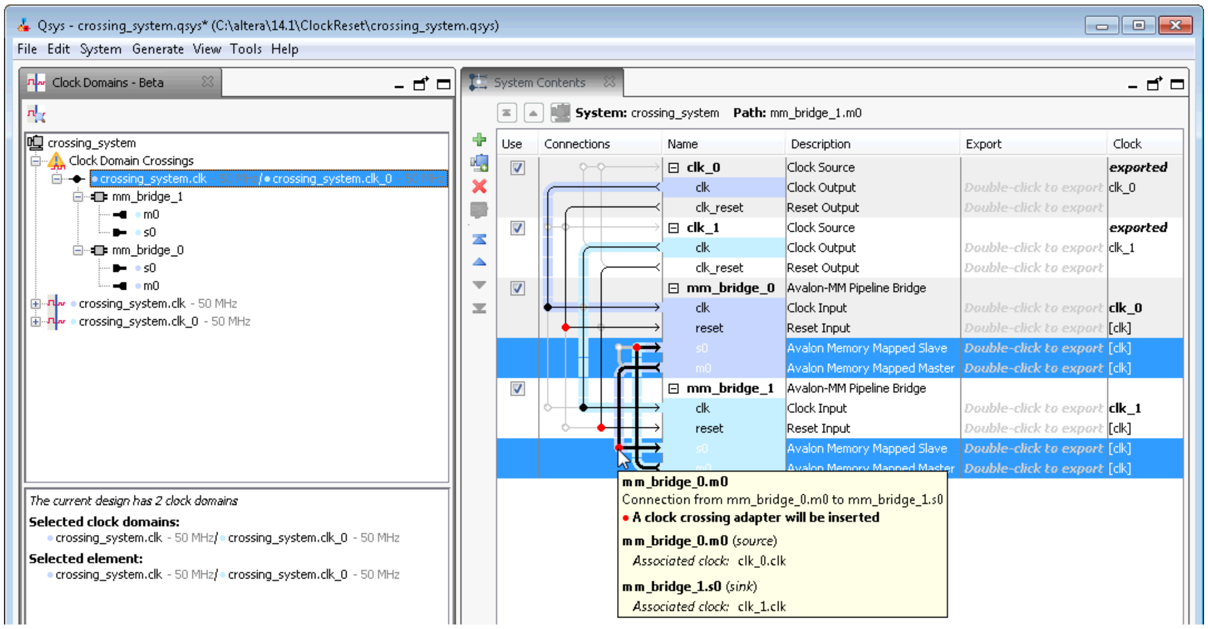
Use	Connections	Name	Description	Export	Clock
✓		**clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	*exported*
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	*Double-click to export*	clk_0
		clk_reset	Reset Output	*Double-click to export*	
✓		**clk_1**	Clock Source		
		clk_in	Clock Input	**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 4-27: 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]

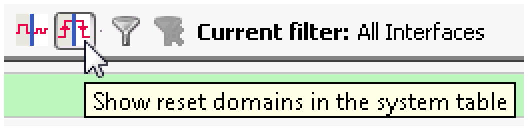
mm_bridge_0.m0
Connection from mm_bridge_0.m0 to mm_bridge_1.s0
● **A clock crossing adapter will be inserted**
mm_bridge_0.m0 *(source)*
Associated clock: clk_0.clk
mm_bridge_1.s0 *(sink)*
Associated clock: clk_1.clk

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 4-28: 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:

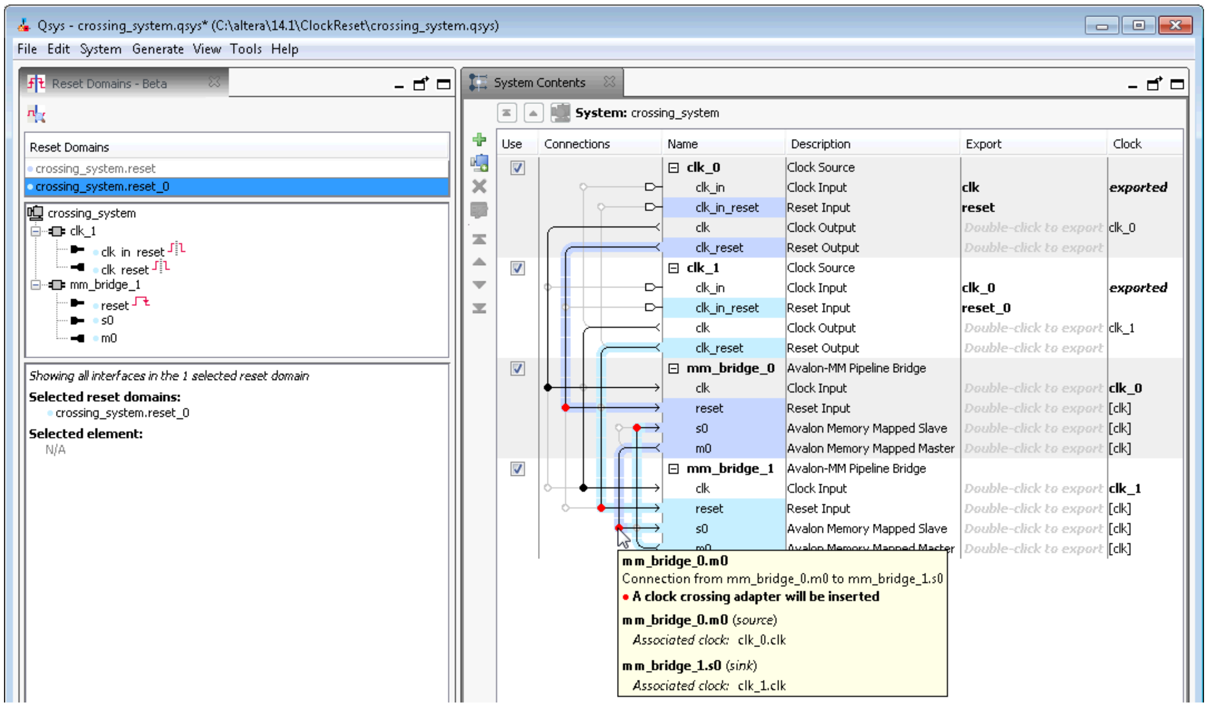
- When you select multiple reset domains, the **System Contents** tab shows interfaces and modules in both reset domains.
- When you select a single reset domain, the other reset domain(s) are grayed out, unless the two domains have interfaces in common.
- Reset interfaces appear black when connected 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 4-29: 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
Use
Connections

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

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]

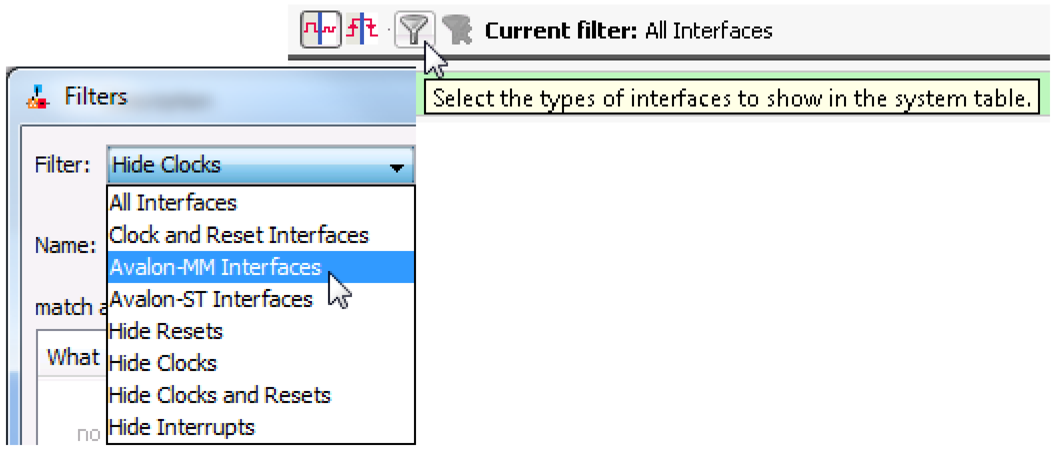
mm_bridge_0.m0
Connection from mm_bridge_0.m0 to mm_bridge_1.s0
● **A clock crossing adapter will be inserted**
mm_bridge_0.m0 (*source*)
Associated clock: clk_0.clk
mm_bridge_1.s0 (*sink*)
Associated clock: clk_1.clk

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.

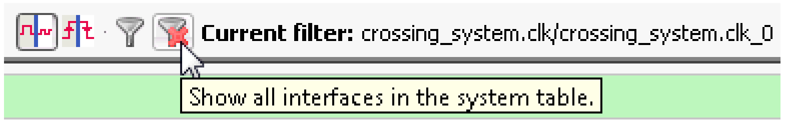
1. To filter the display in the **System Contents** tab to view only a particular interface and its connections, or to choose to hide clock, reset, or interrupt interfaces, click the **Filters** icon in the clock and reset control tool to open the **Filters** dialog box.
   The selected interfaces appear in the **System Contents** tab.

Figure 4-30: 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 4-31: 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 4-5: 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 4-6: 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 6-1
- Qsys System Design Components on page 9-1

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 4-7: 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 4-8: 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 9-1

Integrate a Qsys System with the Quartus Prime Software

To integrate a Qsys system with your Quartus Prime project, you must add either the Qsys System File (**.qsys**) or the Quartus Prime IP File (**.qip**), but never both to your Quartus Prime 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 Prime project.

You can choose to include the **.qsys** file automatically in your Quartus Prime project when you generate your Qsys system by turning on the **Automatically add Quartus Prime IP files to all projects** option in the Quartus Prime software (**Tools** > **Options** > **IP Settings**). If this option is turned off, the Quartus Prime software asks you if you want to include the **.qsys** file in your Quartus Prime project after you exit Qsys.

If you want file generation to occur as part of the Quartus Prime software's compilation, you should include the **.qsys** file in your Quartus Prime project. If you want to manually control file generation outside of the Quartus Prime software, you should include the **.qip** file in your Quartus Prime project.

Note: The Quartus Prime software generates an error message during compilation if you add both the **.qsys** and **.qip** files to your Quartus Prime project.

Does Quartus Prime 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 Prime 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 Prime project directory under **/db/ip**.

For standard devices, Qsys-generated files are only overwritten during Quartus Prime compilation if the **.qip** file is removed or missing. For legacy devices, each time you compile your Quartus Prime 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

- **Introduction to the Qsys IP Catalog** on page 4-3
- **Generate a Qsys System** on page 4-58
- **Qsys Synthesis Standard and Legacy Device Output Directories**
- **Qsys Simulation Standard and Legacy Device Output Directories**
- **Introduction to Altera IP Cores**
- **Implementing and Parameterizing Memory IP**

Integrate a Qsys System and the Quartus Prime Software With the .qsys File

Use the following steps to integrate your Qsys system and your Quartus Prime 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 Prime project during compilation, in the Quartus Prime software, select **Tools** > **Options** > **IP Settings**, and turn on **Automatically add Quartus Prime IP files to all projects**.
3. When the **Automatically add Quartus Prime IP files to all projects** option is not checked, when you exit Qsys, the Quartus Prime software displays a dialog box asking whether you want to add the **.qsys** file to your Quartus Prime project. Click **Yes** to add the **.qsys** file to your Quartus Prime project.
4. In the Quartus Prime software, select **Processing** > **Start Compilation**.

Integrate a Qsys System and the Quartus Prime Software With the .qip File

Use the following steps to integrate your Qsys system and your Quartus Prime project using the **.qip** file:

1. In Qsys, create and save a Qsys system.
2. In Qsys, click **Generate HDL**.
3. In the Quartus Prime 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 Prime project.
5. In the Quartus Prime software, select **Processing** > **Start Compilation**.

Manage IP Settings in the Quartus Prime Software

To specify the following IP Settings in the Quartus Prime software, click **Tools** > **Option** > **IP Settings**:

Table 4-9: 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 Prime 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 Prime IP files to all projects	The Quartus Prime software uses this setting when you create an IP core file variation with options in the Quartus Prime IP Catalog and parameter editor. When turned on, the Quartus Prime software adds the IP variation files to the project currently open.
IP Catalog Search Locations	The Quartus Prime 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 Prime 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 Prime 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 Prime project open. This allows you access to **IP Settings**

when you want to create IP cores independent of a Quartus Prime 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 Prime software **Options** dialog box, or at the command-line.

- When you open Qsys from within the Quartus Prime 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

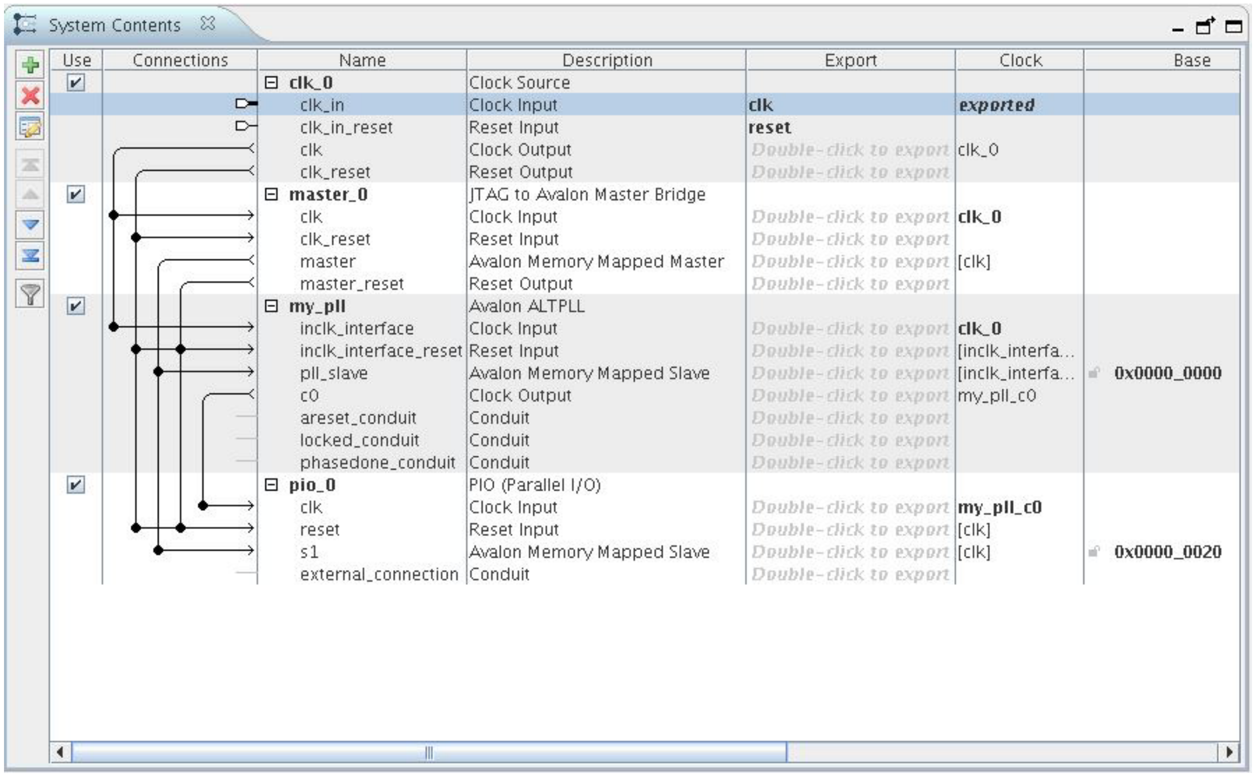
Many IP components include Synopsys Design Constraint (**.sdc**) files that provide timing constraints. Generated **.sdc** files are included in your Quartus Prime project with the generated **.qip** file. For your top-level clocks and PLLs, you must provide clock and timing constraints in SDC format 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 Prime project using the **.sdc** file.

Figure 4-32: 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 Prime 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

Qsys generates files for synthesis in Quartus and simulation in a third-party simulator.

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 3rd party simulator.

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.
Clear output directories for selected generation targets	Clears previous generation attempts for current synthesis or 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.

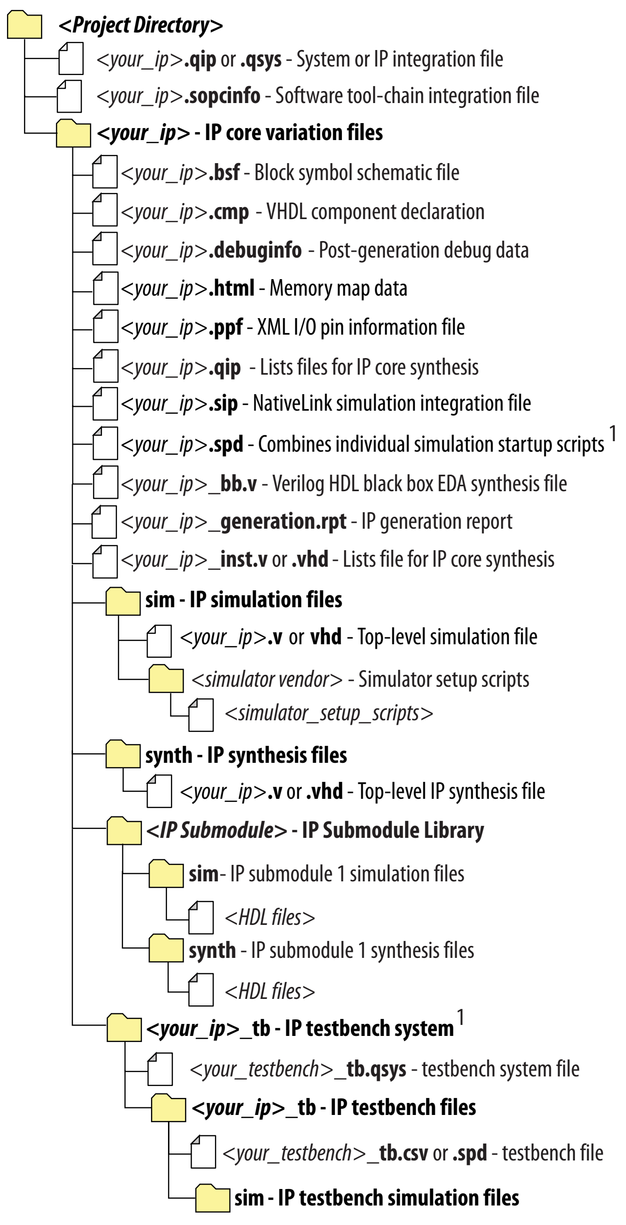
Related Information

Simulating Altera Designs

Files Generated for Altera IP Cores and Qsys Systems

The Quartus Prime software generates the following output file structure for IP cores and Qsys systems. The generated **.qsys** file must be added to your project to represent IP and Qsys systems.

Figure 4-33: Files generated for IP cores and Qsys Systems


<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>.bsf - Block symbol schematic file
<your_ip>.cmp - VHDL component declaration
<your_ip>.debuginfo - Post-generation debug data
<your_ip>.html - Memory map data
<your_ip>.ppf - XML I/O pin information file
<your_ip>.qip - Lists files for IP core synthesis
<your_ip>.sip - NativeLink simulation integration file
<your_ip>.spd - Combines individual simulation startup scripts 1
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_generation.rpt - IP generation report
<your_ip>_inst.v or .vhd - Lists file for IP core synthesis
sim - IP simulation files
<your_ip>.v or vhd - Top-level simulation file
<simulator vendor> - Simulator setup scripts
<simulator_setup_scripts>
synth - IP synthesis files
<your_ip>.v or .vhd - Top-level IP synthesis file
<IP Submodule> - IP Submodule Library
sim- IP submodule 1 simulation files
<HDL files>
synth - IP submodule 1 synthesis files
<HDL files>
<your_ip>_tb - IP testbench system 1
<your_testbench>_tb.qsys - testbench system file
<your_ip>_tb - IP testbench files
<your_testbench>_tb.csv or .spd - testbench file
sim - IP testbench simulation files

1. If supported and enabled for your IP core variation.

Table 4-10: IP Core and Qsys Simulation 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.

File Name	Description
<*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.
<*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 Prime 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 Prime 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*>_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.

File Name	Description
<*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.
<*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.
***<IP submodule>*/**	For each generated IP submodule directory, Qsys generates **/synth** and **/sim** sub-directories.

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 4-11: Qsys-Generated Testbench Files

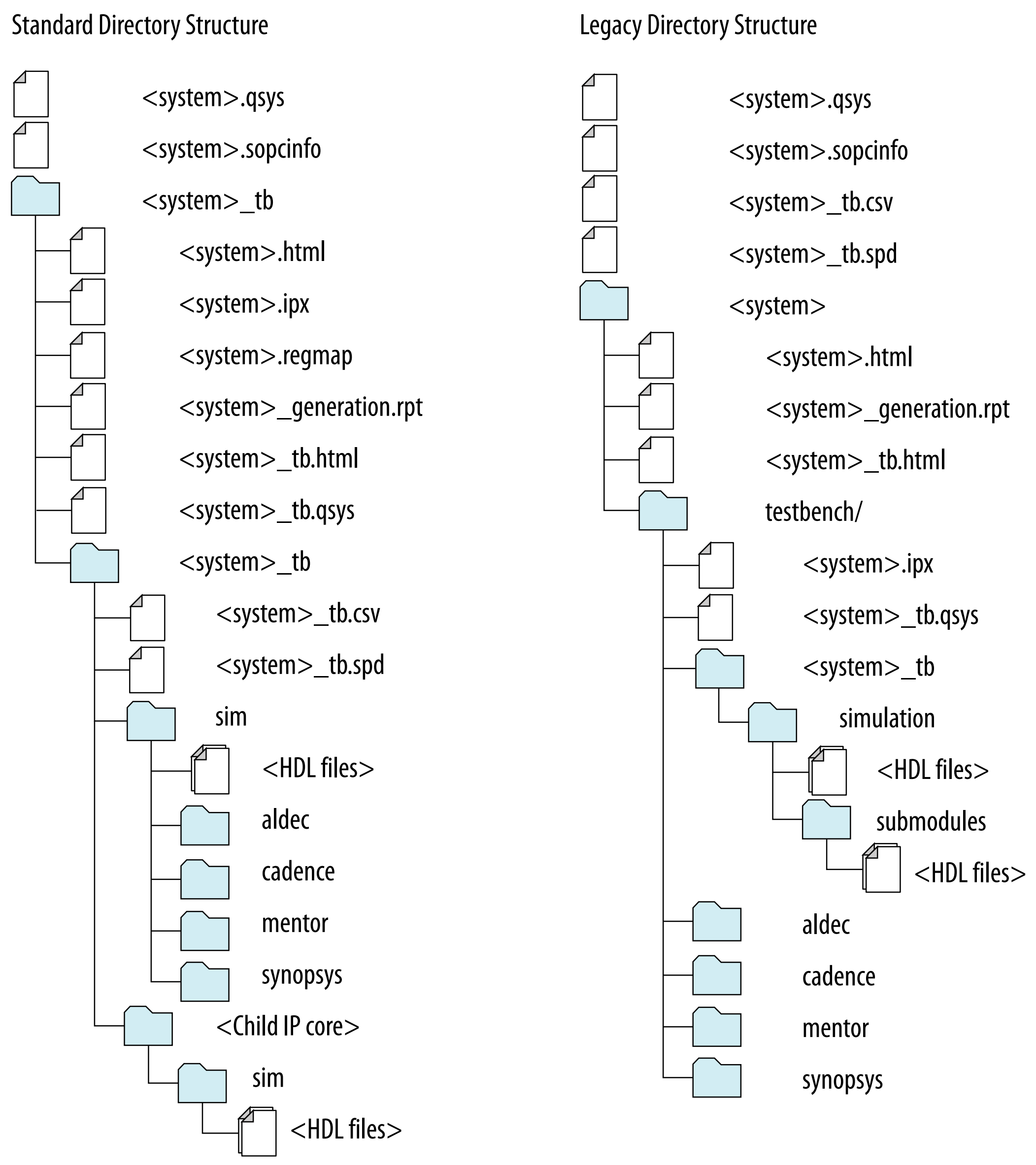
File Name or Directory Name	Description
<*system*>_tb.qsys	The Qsys testbench system.

File Name or Directory Name	Description
<*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.
/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 4-34: 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.

Qsys 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 4-12: 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 4-4: 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 Prime 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

Incorporating IP Simulation Scripts in Top-Level Scripts

Generating Version-Independent IP and Qsys Simulation Scripts

The Quartus Prime software includes useful utilities that generate simulation scripts for each IP core or Qsys system in your design. You can use these utilities to produce a single simulation script that does not require manual update for upgrades to Quartus Prime software or IP versions.

This scripted method generates simulation scripts that support ModelSim-Altera, all supported versions of Questa-SIM, VCS, VCSMX, NCSim, and Aldec simulators. These generated scripts are not suitable for entire design simulation because they lack top-level design information. However, you can easily source the generated scripts from your top-level simulation script. You can incorporate templates from the generated scripts into a top-level script.

Use the `ip-setup-simulation` utility to find all Altera IP cores and Qsys systems in your project. Next, run the `ip-make-simscript` utility to generate a combined IP simulation script. The `ip-setup-simulation` utility also automates regeneration of a combined simulation script following upgrade of the software. If you use simulation scripts, run the `ip-setup-simulation` utility after upgrading software or IP core version.

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.

Table 4-13: IP Simulation Script Utilities

Utility	Description	Syntax	Generated Files
`ip-setup-simulation`	Finds all Altera IP cores in your project and automates regeneration of a combined simulation script after upgrading software or IP versions.	`ip-setup-simulation --quartus-project=<`*`project`*`>.qpf --output-directory=<`*`directory`*`>`	N/A

Utility	Description	Syntax	Generated Files
`ip-make-simscript`	Generates a single, combined simulation script for all of the IP cores specified on the command line. To use `ip-make-simscript`, specify one or more **.spd** files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries. Use the `compileto- work` option to compile all simulation files into a single work library. Use the `--use-relative-paths` option to use relative paths whenever possible	`ip-make-simscript --spd=<`*`ipA.spd,ipB.spd`*`> --output-directory=<directory>`	• Aldec—**aldec/rivierapro_setup.tcl** • Cadence—**cadence/ncsim_setup.sh** • Mentor Graphics—**mentor/msim_setup.tcl** • Synopsys—**synopsys/vcs/vcs_setup.sh**

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)

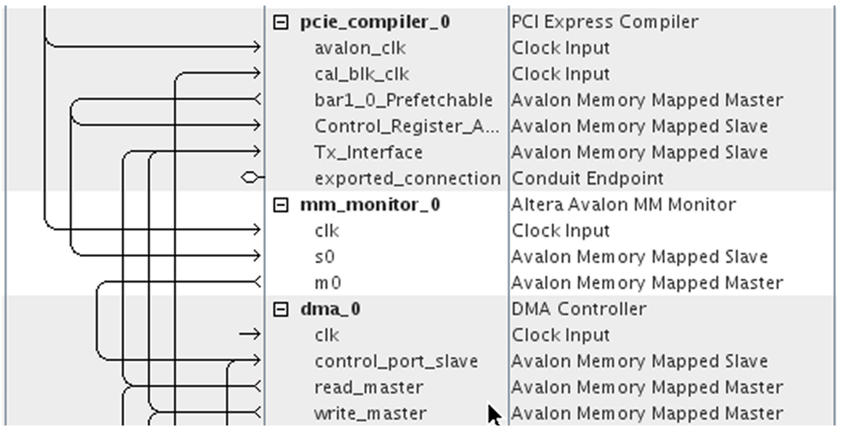
Add Assertion Monitors for Simulation

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 4-35: 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 8-1

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 Prime 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 9-1

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 Prime 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 Prime 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 Prime 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 Prime Handbook*.

Related Information

- **Qsys System Design Components** on page 9-1

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 Prime 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 Prime installation directory:

<*Quartus Prime 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 Prime installation directory*>**\quartus\sopc_builder\bin**\<*executable name*> --help

Example 4-5: 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

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 4-14: 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 use the `qsys-script` and `qsys-generate` utilities to define and generate an IP core variation outside of the Quartus Prime GUI.

To parameterize and generate an IP core at the command-line, follow these steps:

1. Run `qsys-script` to execute a Tcl script that instantiates the IP and sets desired parameters:

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

2. Run `qsys-generate` to generate the IP core 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 4-15: 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 Prime 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.
`--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.

Option	Usage	Description
`--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.

For command-line help listing all options for these executables, type <*executable name*> `--help`

Display Available IP Components with ip-catalog

The `ip-catalog` command displays a list of available IP components relative to the current Quartus Prime project directory. Use the following format for the `ip-catalog` command:

```
ip-catalog
            [--project-dir=<directory>]
   [--name=<value>]
   [--verbose]
   [--xml]
   [--help]
```

Table 4-16: ip-catalog Command-Line Options

Option	Usage	Description
`--project-dir= <directory>`	Optional	Finds IP components relative to the Quartus Prime 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.

Option	Usage	Description
`--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 4-17: 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.
`--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 4-6

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 4-18: 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 Prime 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.
`--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:

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 4-102
- **get_connection_property** on page 4-105
- **get_connections** on page 4-106
- **remove_connection** on page 4-142
- **set_connection_parameter_value** on page 4-148

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 4-121
- **get_instance_property** on page 4-125
- **get_instances** on page 4-126
- **remove_instance** on page 4-144
- **set_instance_parameter_value** on page 4-149

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 4-130
- **get_interface_properties** on page 4-131
- **get_interface_property** on page 4-132
- **set_interface_property** on page 4-152

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 4-88
- **lock_avalon_base_address** on page 4-141
- **unlock_avalon_base_address** on page 4-157

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 4-87
- **lock_avalon_base_address** on page 4-141
- **unlock_avalon_base_address** on page 4-157

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 4-83

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 4-140
- **save_system** on page 4-146

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 4-94
- get_composed_connections on page 4-95

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 4-93
- **get_composed_connections** on page 4-95

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 4-93
- **get_composed_connection_parameters** on page 4-94

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 4-97
- **get_composed_instances** on page 4-100

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 4-96
- **get_composed_instances** on page 4-100

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 4-99
- get_composed_instances on page 4-100

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 parameter 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 4-98
- **get_composed_instances** on page 4-100

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 4-96
- **get_composed_instance_assignments** on page 4-97
- **get_composed_instance_parameter_value** on page 4-98
- **get_composed_instance_parameters** on page 4-99

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 4-102
- **get_connection_property** on page 4-105
- **get_connections** on page 4-106
- **get_parameter_properties** on page 4-136
- **Parameter Properties** on page 4-171

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 4-103
- **get_connections** on page 4-106
- **set_connection_parameter_value** on page 4-148

get_connection_parameters

Description

Returns a list of parameters found on a connection.

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

- **get_connection_parameter_property** on page 4-101
- **get_connection_parameter_value** on page 4-102
- **get_connection_property** on page 4-105

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 4-105
- get_connections on page 4-106

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 4-104
- Connection Properties on page 4-163

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 4-83
- **remove_connection** on page 4-142

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 4-108

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 4-107

get_instance_assignment on page 4-107

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 4-111

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 4-110

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 4-114
- **get_instance_interfaces** on page 4-119
- **get_parameter_properties** on page 4-136
- **Parameter Properties** on page 4-171

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 4-114
- get_instance_interfaces on page 4-119

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 4-113
- **get_instance_interfaces** on page 4-119

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 4-116
- **get_port_properties** on page 4-137
- **Port Properties** on page 4-176

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 4-115
- **get_instance_interfaces** on page 4-119

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 4-118
- get_instance_interfaces on page 4-119

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 4-117
- get_instance_interfaces on page 4-119
- Element Properties on page 4-166

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 4-116
- **get_instance_interface_properties** on page 4-117
- **get_instance_interface_property** on page 4-118

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 4-122
- get_parameter_properties on page 4-136
- Parameter Properties on page 4-171

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 4-122
- **set_instance_parameter_value** on page 4-149

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 4-120
- **get_instance_interface_parameter_value** on page 4-113
- **set_instance_parameter_value** on page 4-149

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 4-116
- **get_port_properties** on page 4-137
- **Port Properties** on page 4-176

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 4-125

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 4-124
- **Element Properties** on page 4-166

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 4-84
- remove_instance on page 4-144

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 4-130
- **get_port_properties** on page 4-137
- **Port Properties** on page 4-176

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 4-129
- **get_interfaces** on page 4-133

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 4-132
- set_interface_property on page 4-152

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 4-131
- set_interface_property on page 4-152
- Interface Properties on page 4-168

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 4-85
- **get_interface_ports** on page 4-130
- **get_interface_property** on page 4-132
- **remove_interface** on page 4-145
- **set_interface_property** on page 4-152

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 4-135
- **set_module_property** on page 4-153

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 4-134
- **set_module_property** on page 4-153

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 4-101
- get_instance_interface_parameter_property on page 4-112
- get_instance_parameter_property on page 4-120

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 4-115
- **get_instance_interface_ports** on page 4-116
- **get_instance_port_property** on page 4-123
- **get_interface_port_property** on page 4-129
- **get_interface_ports** on page 4-130

get_project_properties

Description

Returns a list of properties that you can query for properties pertaining to the Quartus Prime 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 4-139
- **set_project_property** on page 4-154

get_project_property

Description

Returns the value of a Quartus Prime project property. Not all Quartus Prime 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 4-134
- **get_module_property** on page 4-135
- **set_module_property** on page 4-153

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 4-91
- **save_system** on page 4-146

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 4-87
- **auto_assign_system_base_addresses** on page 4-88
- **unlock_avalon_base_address** on page 4-157

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 4-83
- **get_connections** on page 4-106

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 4-84
- **get_instances** on page 4-126

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 4-85
- **get_interfaces** on page 4-133

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 4-169

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 4-102
- **get_connection_parameters** on page 4-103

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 4-121
- **get_instance_parameter_property** on page 4-120

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 4-122
- **get_instance_property** on page 4-125
- **Instance Properties** on page 4-167

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 4-85
- **get_interface_properties** on page 4-131
- **get_interface_property** on page 4-132
- **Interface Properties** on page 4-168

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 4-134
- **get_module_property** on page 4-135
- **Module Properties** on page 4-170

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 4-138
- **set_project_property** on page 4-154
- **Project Properties** on page 4-177
- **get_project_properties** on page 4-138
- **get_project_property** on page 4-139
- **Project Properties** on page 4-177

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 4-161
- Validation Properties on page 4-181

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 4-87
- auto_assign_system_base_addresses on page 4-88
- lock_avalon_base_address on page 4-141

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 4-159
- **validate_instance_interface** on page 4-160
- **validate_system** on page 4-161

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 4-158
- **validate_instance_interface** on page 4-160
- **validate_system** on page 4-161

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 4-158
- validate_instance on page 4-159
- validate_system on page 4-161

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 4-158
- validate_instance on page 4-159
- validate_instance_interface on page 4-160

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 4-178
- **Parameter Type Properties** on page 4-175
- **Units Properties** on page 4-180

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 Prime project that contains the Qsys system.
String	`DEVICE_FAMILY`	The device family name in the Quartus Prime 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 4-164
- **Avalon Interface Specifications**
- **Qsys Interconnect** on page 6-1

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 4-19: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Added: *Troubleshooting IP or Qsys System Upgrade.* • Added: *Generating Version-Agnostic IP and Qsys Simulation Scripts.* • Changed instances of *Quartus II* to *Quartus Prime.*
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime.*
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 Prime 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 Prime Software.* • Added *Upgrading Outdated IP Components.* • Added *Support for Avalon-MM Non-Power of Two Data Widths.*

Date	Version	Changes
November 2013	13.1.0	• Added *Integrating with the .qsys File*. • Added *Using the Hierarchy Tab*. • Added *Managing Interconnect Requirements*. • Added *Viewing Qsys Interconnect*.
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 Handbook Archive

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

Creating Qsys Components 5

2015.11.02

QPP5V1 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 Prime 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.

Interface Support in Qsys

IP components (IP Cores) 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.

Component Structure

Altera provides components automatically installed with the Quartus Prime 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 5-28
- **Add Component Instances to a Static or Generated Component** on page 5-31
- **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 Prime 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 allows you to create and package an IP component. When you use the Component Editor to define a component, Qsys writes the information to an **_hw.tcl** file.

The Qsys 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, VHDL files, and constraint files that define the component for synthesis and simulation.
- Create an HDL template to define a component interfaces, signals, and parameters.
- Set parameters on interfaces and signals that can alter the component's structure or functionality.

If you add the top-level HDL file that defines the component on **Files** tab in the Qsys Component Editor, you must define the component's parameters and signals in the HDL file. You cannot add or remove them in the Component Editor.

If you do not have a top-level HDL component file, you can use the Qsys Component Editor to add interfaces, signals, and parameters. In the Component Editor, the order in which the tabs appear reflects the recommended design flow for component development. You can use the **Prev** and **Next** buttons to guide you through the tabs.

In a Qsys system, the interfaces of a component are connected in the system, or exported as top-level signals from the system.

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 that you enter in the Component Editor.

When you save the component, it appears in the IP Catalog.

If you require custom features that the Qsys Component Editor does not support, for example, an elaboration callback, use the Component Editor to create the **_hw.tcl** file, and then manually edit the file to complete the component definition.

Note: If you add custom coding to a component, do not open the component file in the Qsys Component Editor. The Qsys Component Editor overwrites your custom edits.

Example 5-1: 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 8-1

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 move **_hw.tcl** files and their associated files to an **ip/** directory within your Quartus Prime project directory. You can use IP components 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 4-1

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 Prime 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.

Specify IP Component Type Information

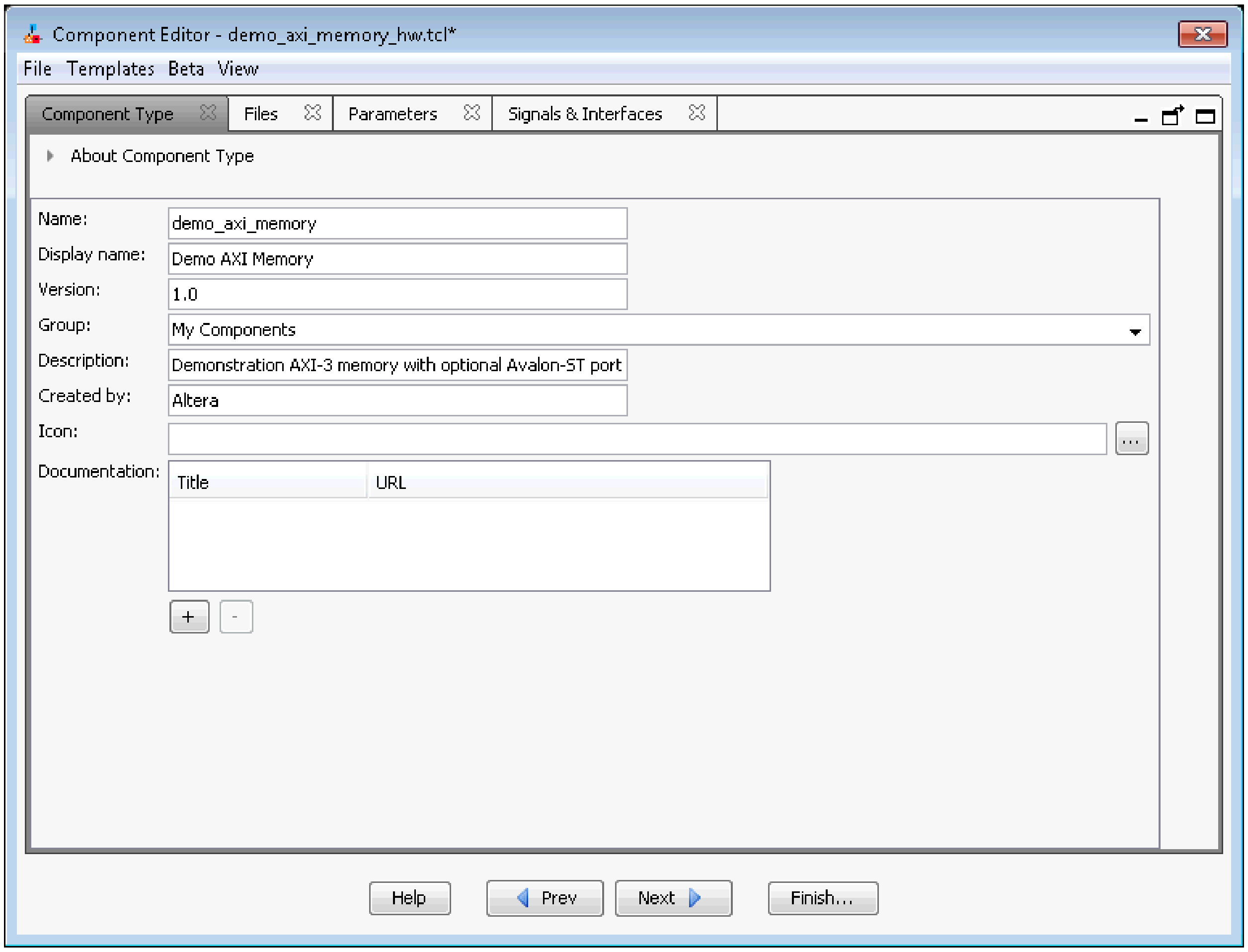
The **Component Type** tab in the Qsys Component Editor allows you to specify the following information about the component:

- **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 Prime 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 5-1: 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:	

When you use the Component Editor to create a component, it writes this basic component information in the **_hw.tcl file**. The `package require` command specifies the Quartus Prime 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 5-2: _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 Prime 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 8-1

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

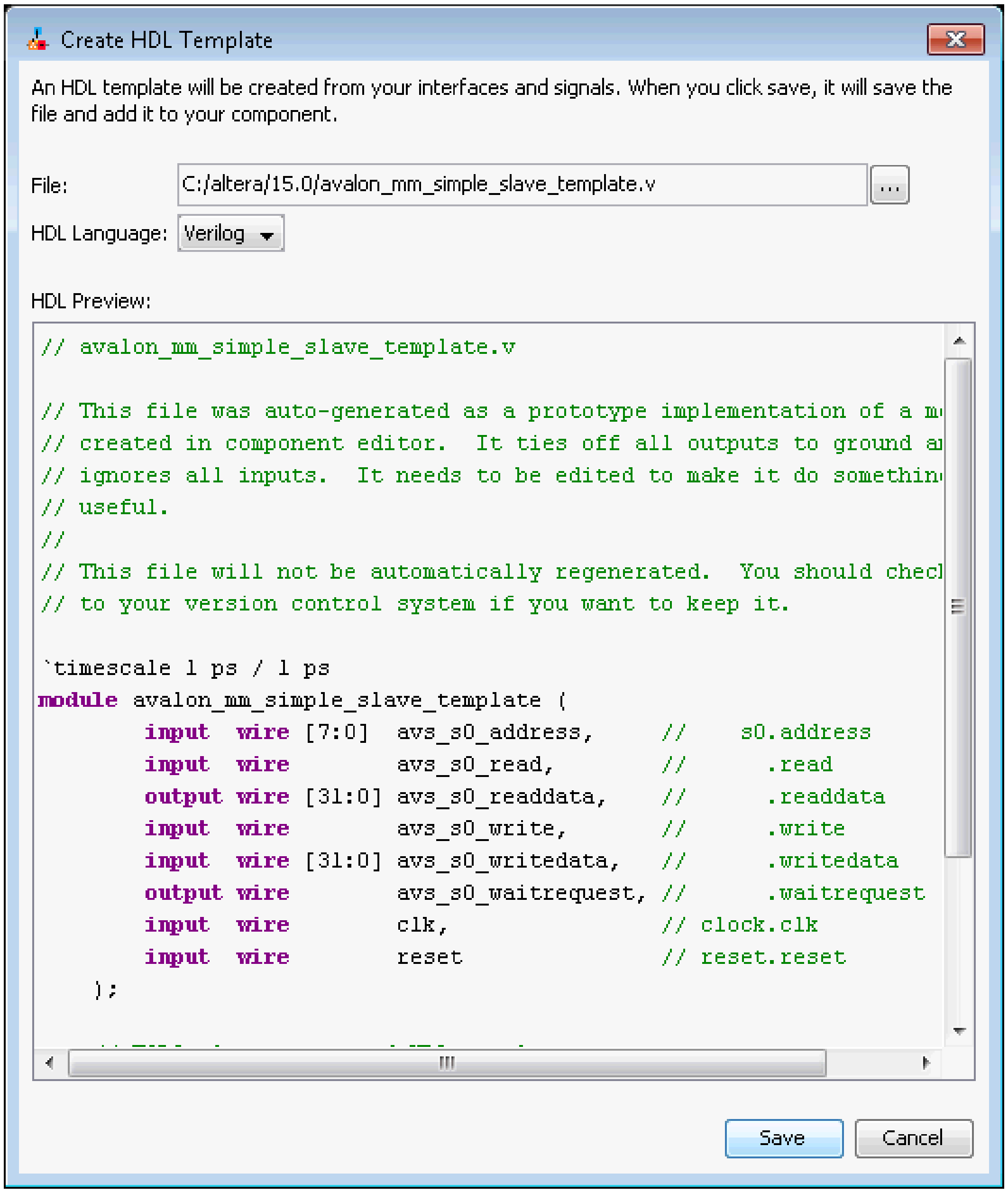
Specify Synthesis and Simulation Files in the Qsys Component Editor on page 5-11

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 5-11

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 Prime 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 5-9
- Create an HDL File Using a Template in the Qsys Component Editor on page 5-9

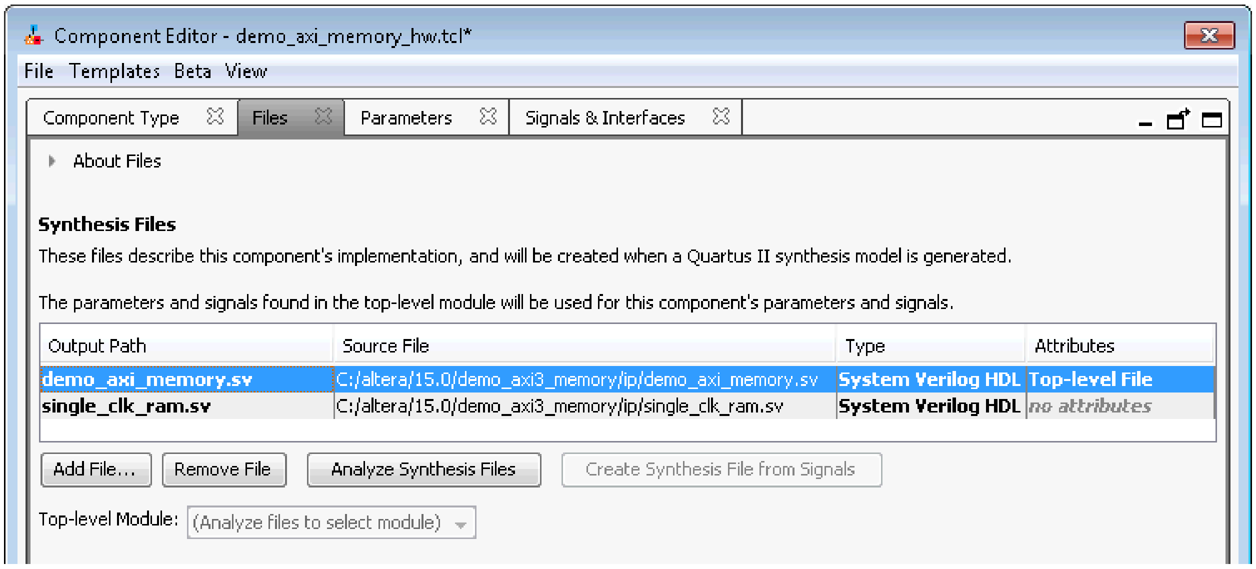
Specify HDL Files for Synthesis in the Qsys Component Editor

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 Prime software. The synthesis files for a component are copied to the generation output directory during Qsys system generation.

Figure 5-2: 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

Add File... | Remove File | Analyze Synthesis Files | Create Synthesis File from Signals

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 5-3: _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 5-12

Component Interface Tcl Reference on page 8-1

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 5-2: 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

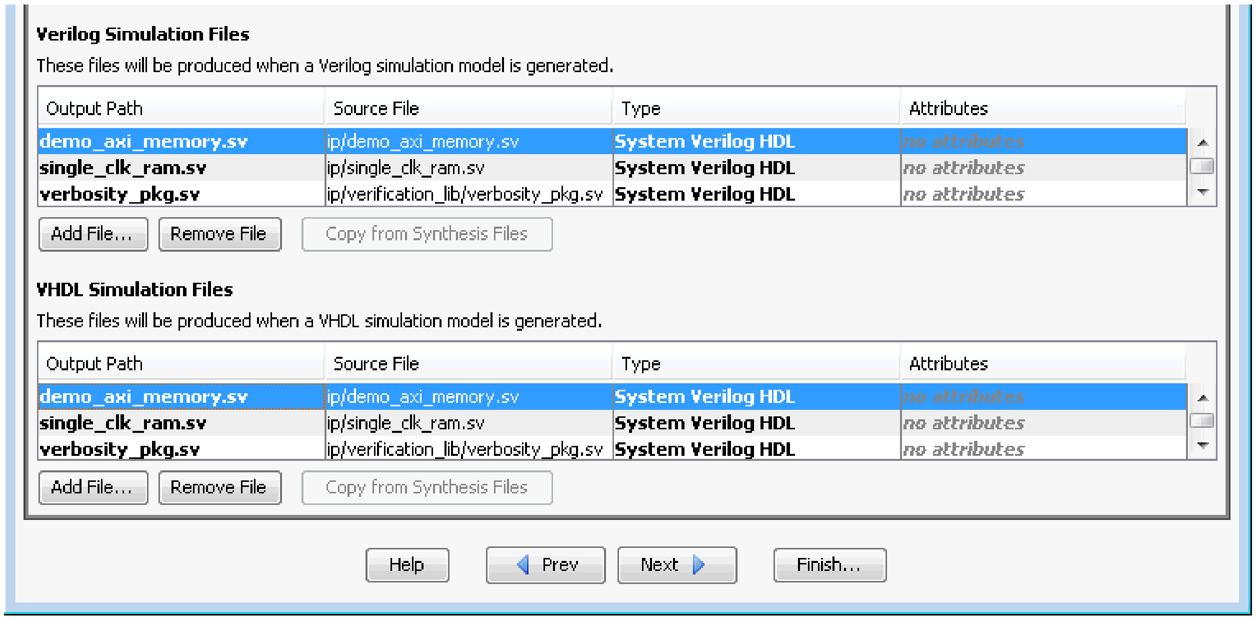
Specify Files for Simulation in the Component Editor

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 5-3: 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 5-4: _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 8-1

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 5-5: 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 8-1

CMSIS - Cortex Microcontroller Software

Add Signals and Interfaces in the Qsys Component Editor

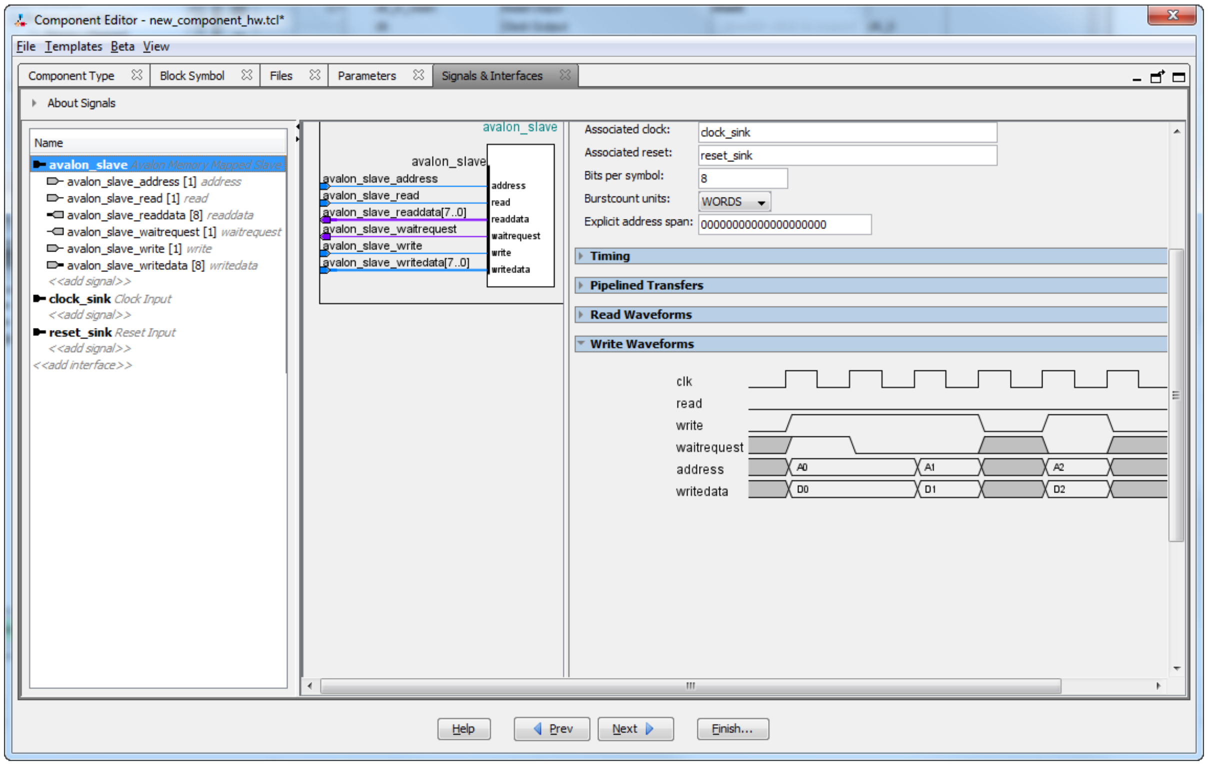
In the Qsys Component Editor, the **Signals & Interfaces** tab allows you to add signals and interfaces for your custom IP component.

As you select interfaces and associated signals, you can customize the parameters. Messages appear as you add interfaces and signals to guide you when customizing the component. In the parameter editor, a block diagram displays for each interface. Some interfaces display waveforms to show the timing of the interface. If you update timing parameters, the waveforms update automatically.

1. In Qsys, click **New Component** in the IP Catalog.
2. In the Qsys Component Editor, click the **Signals & Interfaces** tab.
3. To add an interface, click <<*add interface*>> in the left pane.
   A drop-down list appears where you select the interface type.
4. Select an interface from the drop-down list.
   The selected interface appears in the parameter editor where you can specify its parameters.
5. To add signals for the selected interface click <<*add signal*>> below the selected interface.

6. To move signals between interfaces, select the signal, and then drag it to another interface.
7. To rename a nsignal or interface, select the element, and then press **F2**.
8. To remove a signal or interface, right-click the element, and then click **Remove**.
   Alternatively, to remove an signal or interface, you can select the element, and then press **Delete**.
   When you remove an interface, Qsys also removes all of its associated signals.

Figure 5-4: Qsys Signals & Interfaces tab



avalon_slave Avalon Memory Mapped Slave
avalon_slave_address [1] address
avalon_slave_read [1] read
avalon_slave_readdata [8] readdata
avalon_slave_waitrequest [1] waitrequest
avalon_slave_write [1] write
avalon_slave_writedata [8] writedata
<<add signal>>
clock_sink Clock Input
<<add signal>>
reset_sink Reset Input
<<add signal>>
<<add interface>>

avalon_slave_address
avalon_slave_read
avalon_slave_readdata[7..0]
avalon_slave_waitrequest
avalon_slave_write
avalon_slave_writedata[7..0]

Associated clock: clock_sink
Associated reset: reset_sink
Bits per symbol: 8
Burstcount units: WORDS
Explicit address span: 00000000000000000000

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.

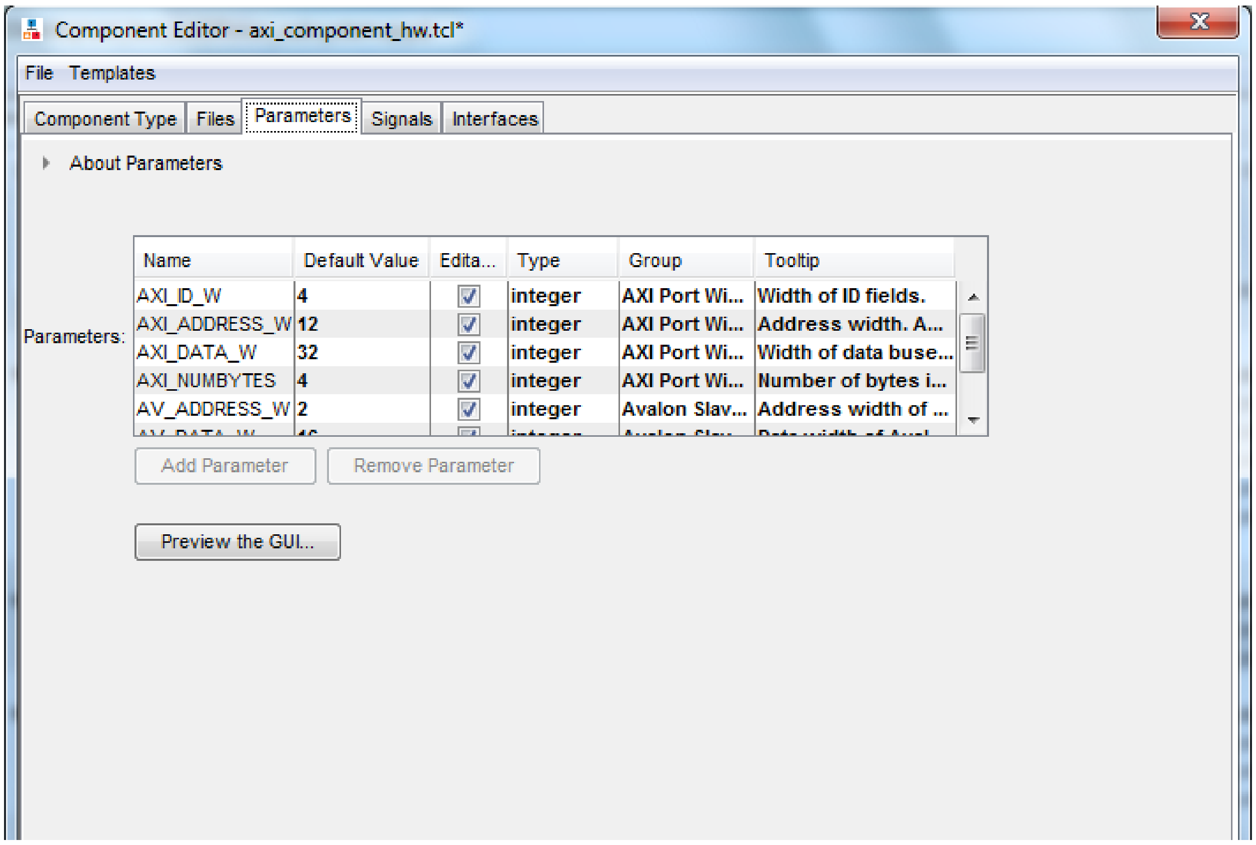
If you used the Component Editor to create a top-level template HDL file for synthesis, you can remove the newly-created file from the **Synthesis Files** list on the **Files** tab, make your parameter changes, and then re-analyze the top-level synthesis file.

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 5-5: 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 5-6: _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 5-3: 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 8-1

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 5-4: 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 5-23

Types of Qsys Parameters

Qsys uses the following parameter types: user parameters, system information parameters, and derived parameters.

Qsys User Parameters on page 5-22

Qsys System Information Parameters on page 5-23

Qsys Derived Parameters on page 5-23

Related Information

Declare Parameters with Custom _hw.tcl Commands on page 5-23

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 5-23

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.

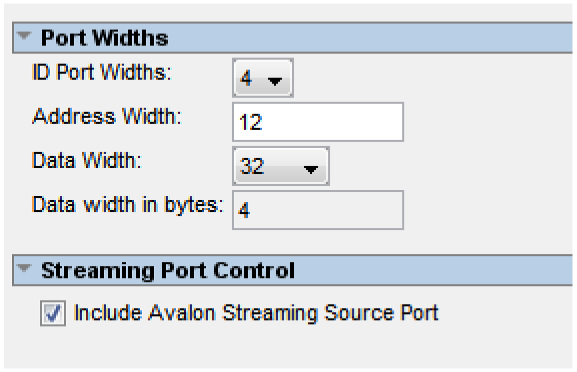
Example 5-7: Parameter Declaration

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 5-6: Resulting Parameter Editor GUI from Parameter Declarations


Port Widths

ID Port Widths: 4

Address Width: 12

Data Width: 32

Data width in bytes: 4

Streaming Port Control

☑ Include Avalon Streaming Source Port

Related Information

Control Interfaces Dynamically with an Elaboration Callback on page 5-26

Component Interface Tcl Reference on page 8-1

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 5-8: 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 8-1

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 5-9: 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 5-25
- **Component Interface Tcl Reference** on page 8-1

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 5-10: 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 5-11

Component Interface Tcl Reference on page 8-1

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.

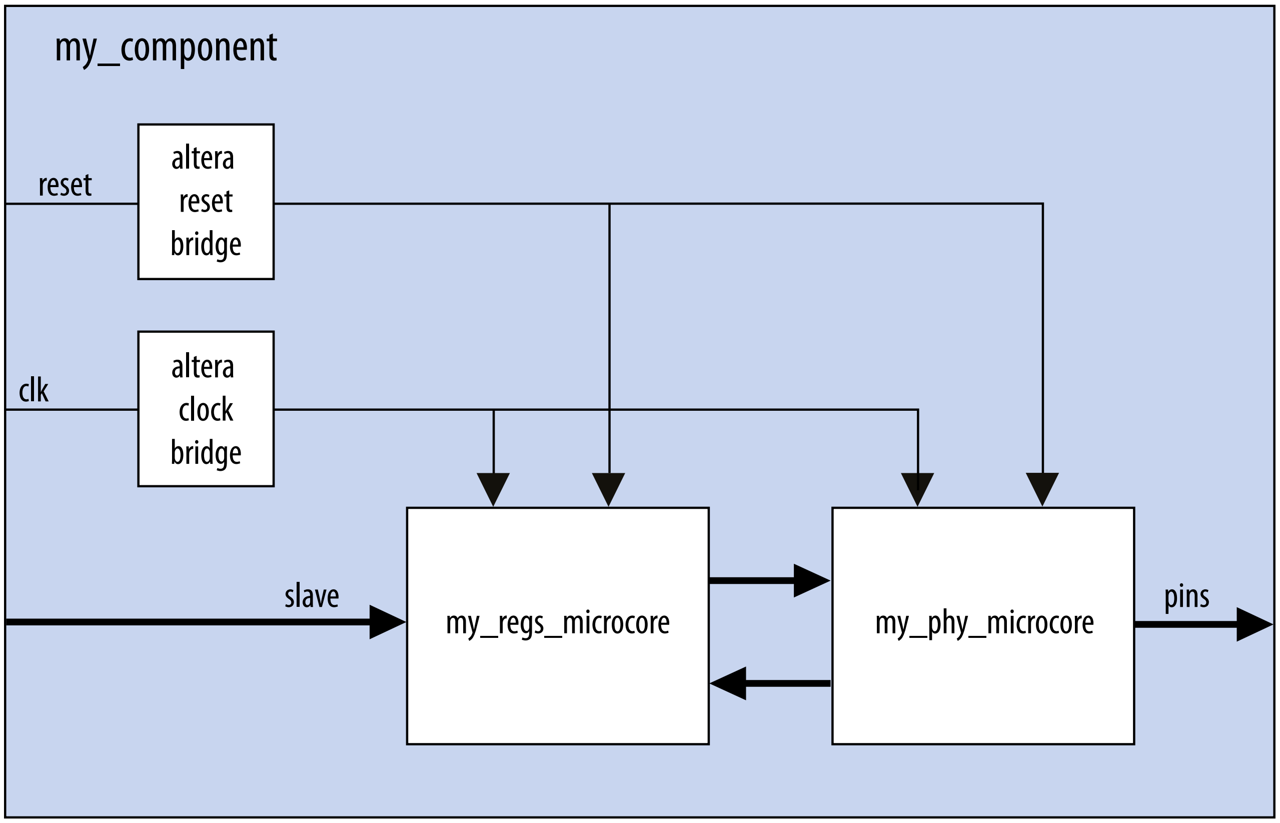
To connect instances of your component, you must define the component's interfaces. Unlike an HDL-based component, a composed component does not directly specify the signals that are exported. Instead, interfaces of submodules are chosen as the external interface, and each internal interface's ports 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 5-7: Top-Level of a Composed Component



Example 5-11: 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 8-1

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 Prime 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 5-12: 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 5-28

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 5-32
- **Generated Components** on page 5-33

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 5-13: 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 5-14: 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 5-15: 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 5-16: 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 5-26

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 5-5: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
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*.

Date	Version	Changes
November 2013	13.1.0	• `add_hdl_instance` • Added *Creating a Component With Differing Structural Qsys View and Generated Output Files*.
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.

Related Information

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

Qsys Interconnect 6

2015.11.02

QPP5V1 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 4-1
- **Creating Qsys Components** on page 5-1
- **Qsys System Design Components** on page 9-1
- **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.

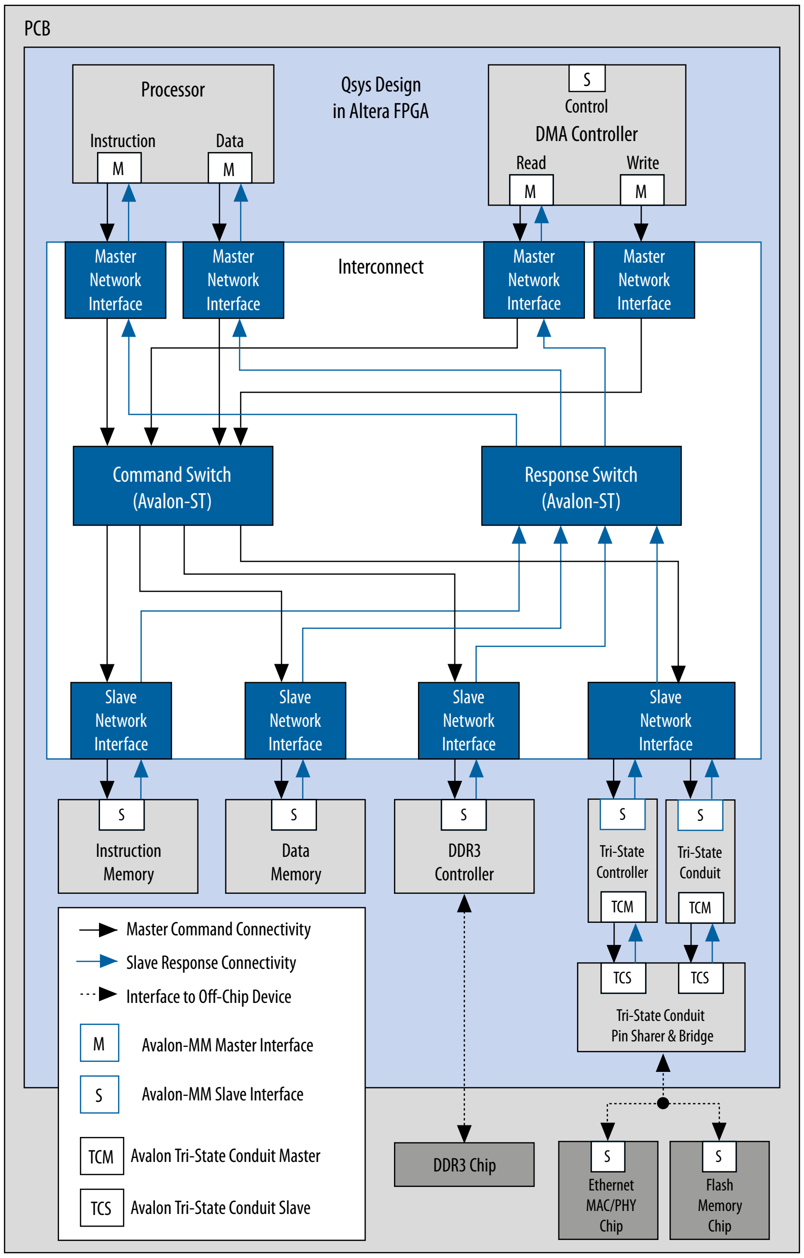
Qsys interconnect supports the following implementation scenarios:

- 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 6-1: 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 Prime 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 6-1: 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 6-2: Transaction Types for Memory-Mapped Interfaces

The table below describes the information that each bit transports in the packet format's transaction field.

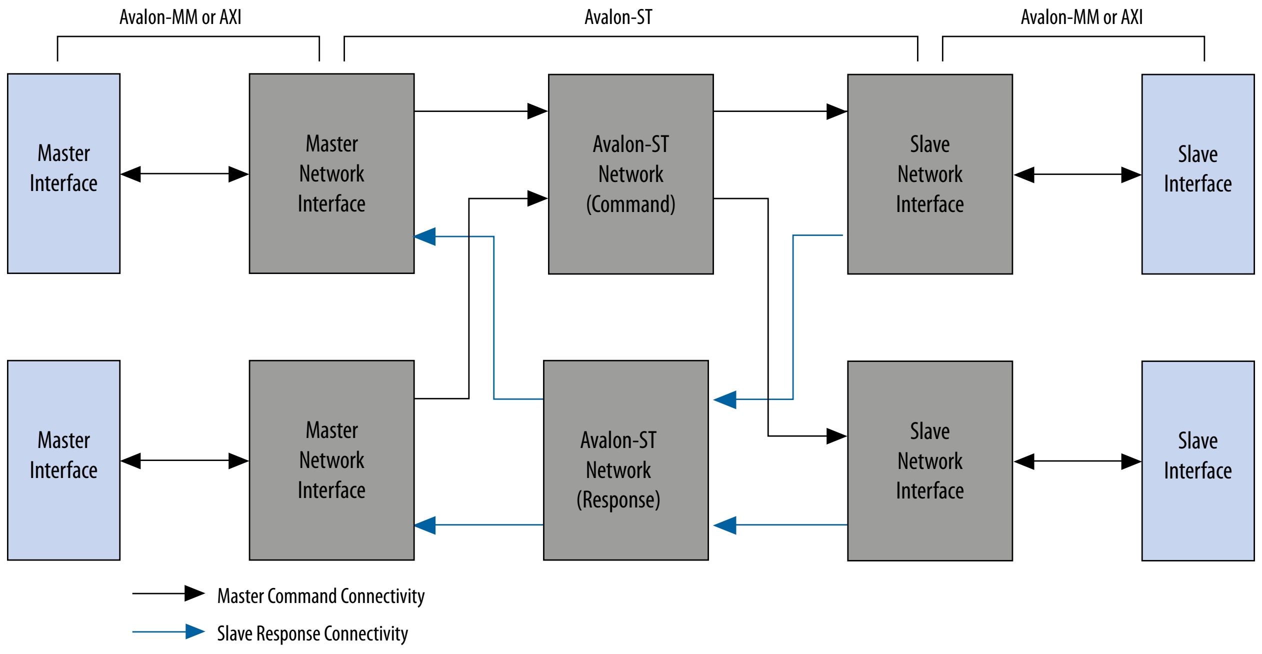
Bit	Name	Definition
0	`PKT_TRANS_READ`	When asserted, indicates a read transaction.
1	`PKT_TRANS_COMPRESSED_READ`	For read transactions, specifies whether or not the read command can be expressed in a single cycle, that is whether or not it has all `byteenables` asserted on every cycle.
2	`PKT_TRANS_WRITE`	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 6-2: 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 6-11
- **Slave Network Interfaces** on page 6-13

Interconnect Domains

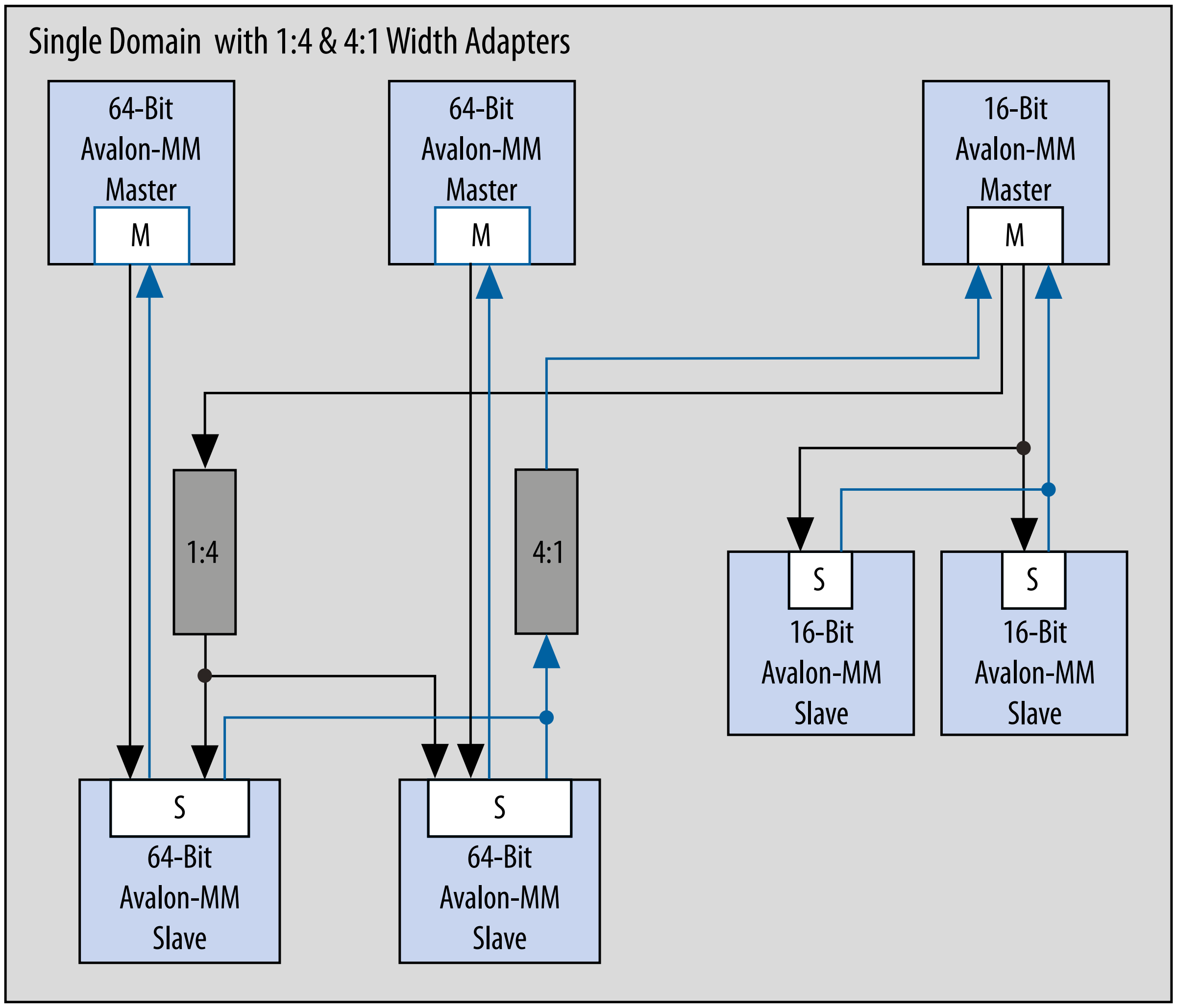
An interconnect domain is a group of connected memory-mapped masters and slaves that share the same interconnect. The components in a single interconnect domain share the same packet format.

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 6-3: One Domain with 1:4 and 4:1 Width Adapters

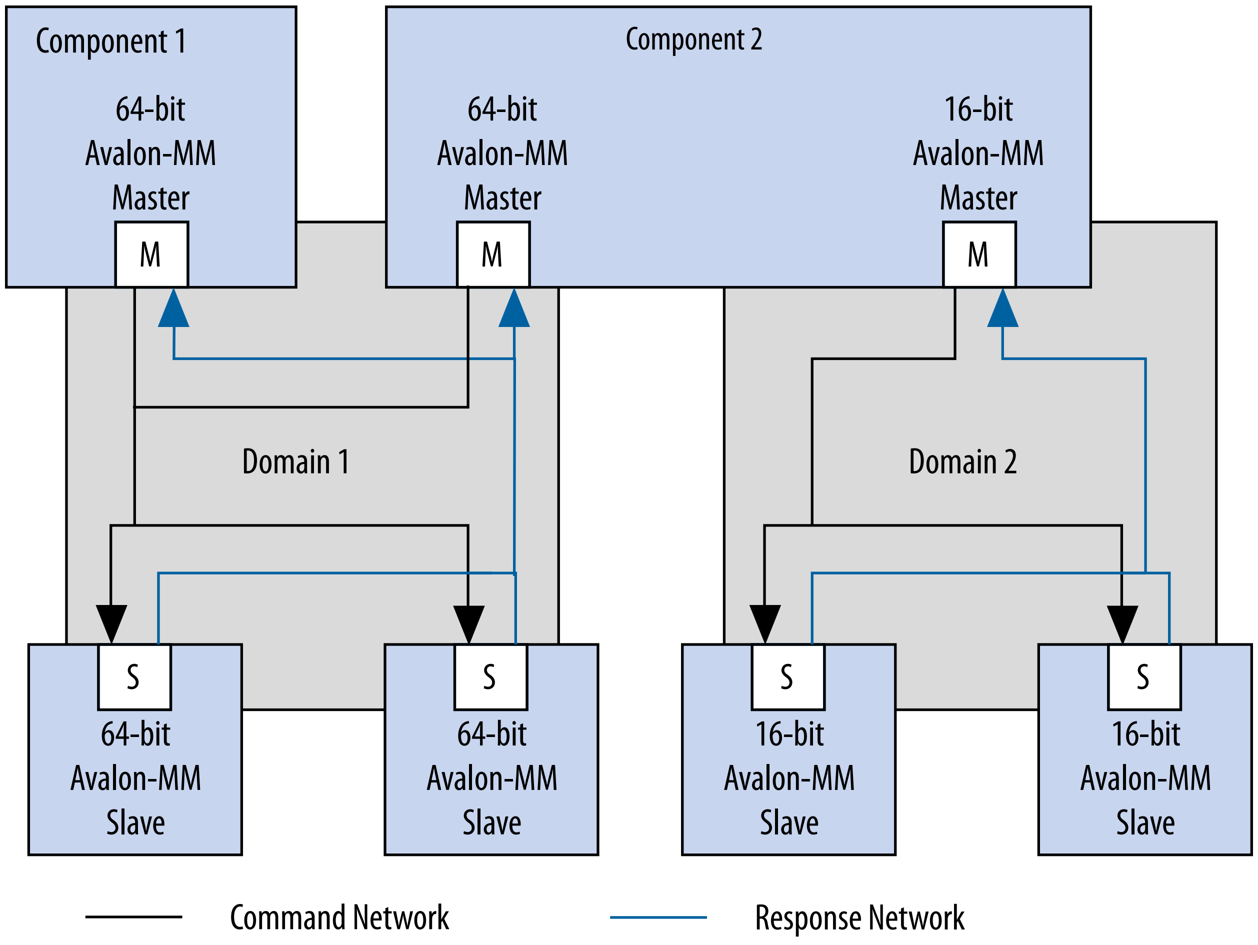
In this system example, there are two 64-bit masters that access two 64-bit slaves. It also includes one 16-bit master, that accesses two 16-bit slaves and two 64-bit slaves. The 16-bit Avalon master connects through a 1:4 adapter, then a 4:1 adapter to reach its 16-bit slaves.



Using Two Separate Domains

Figure 6-4: 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 6-5: 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.

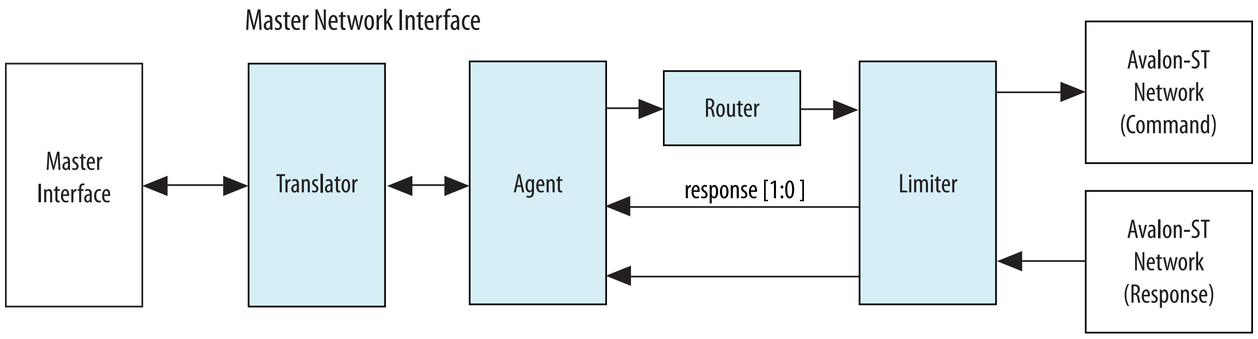
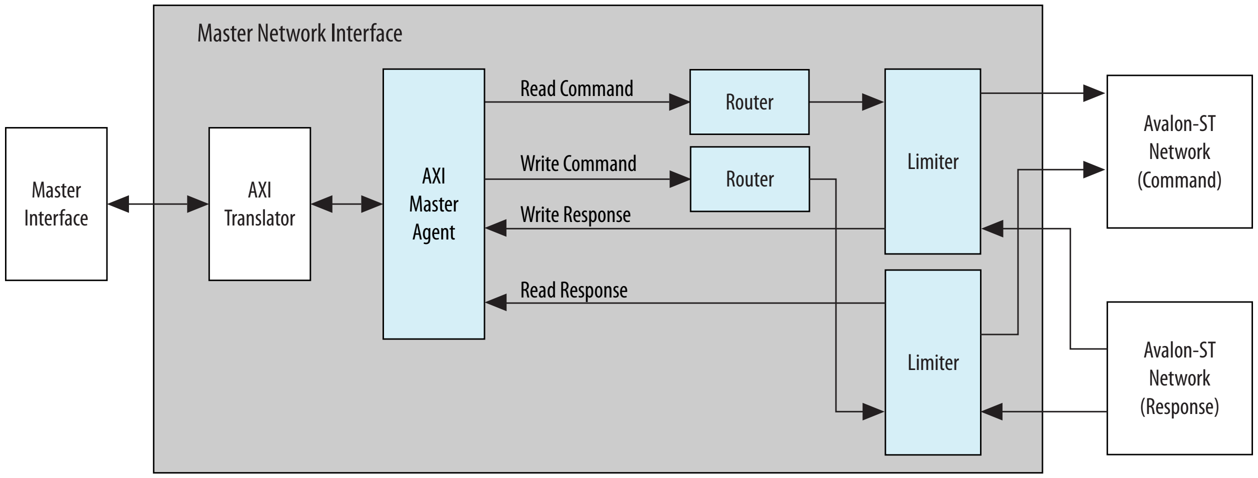


Figure 6-6: AXI Master Network Interface

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 6-27
- Avalon Interface Specifications
- Creating a System with Qsys on page 4-1

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 6-4

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 6-7: Avalon-MM Slave Network Interface

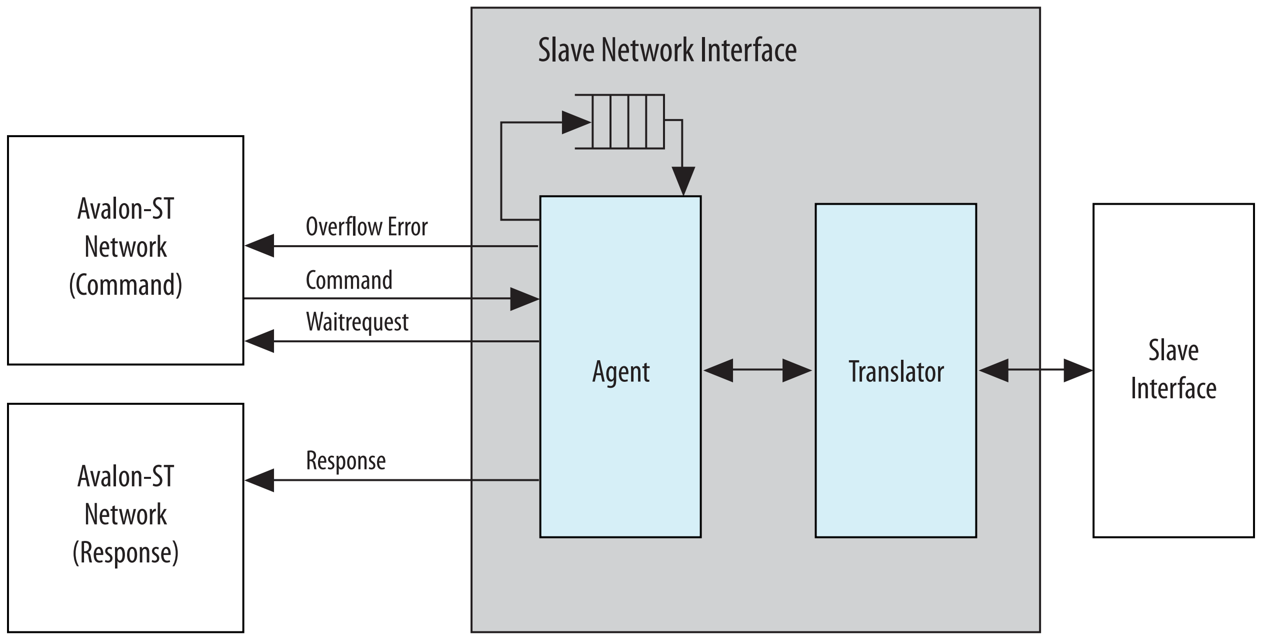
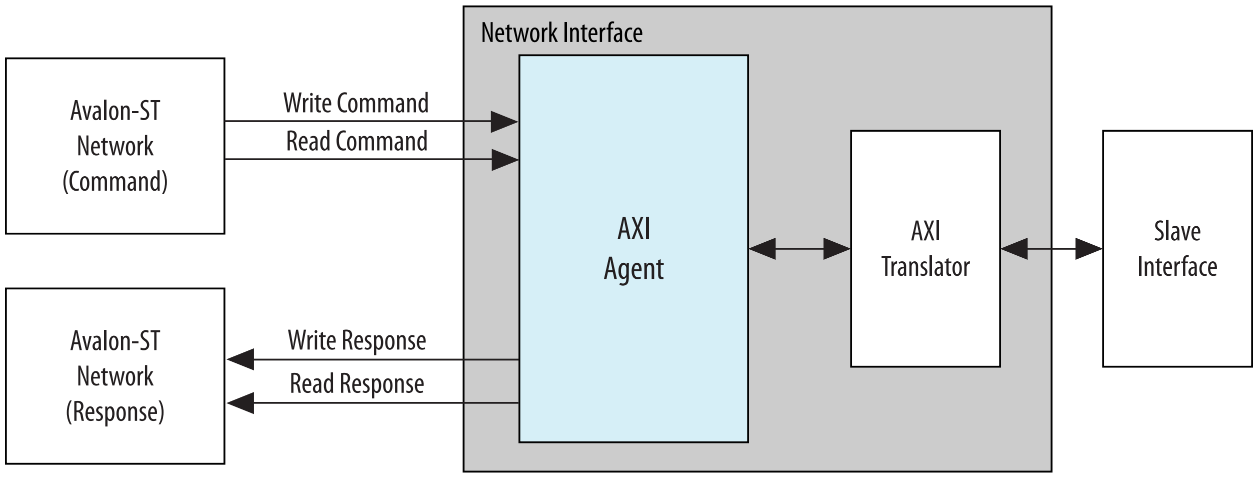


Figure 6-8: AXI Slave Network Interface

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 6-13

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:

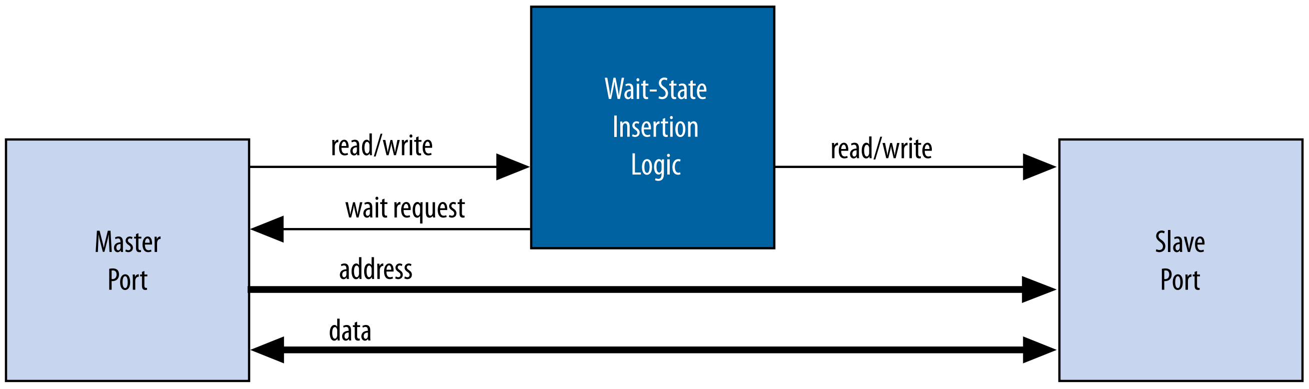
- Matches ID widths between master and slave in 1x1 systems.
- Drives default values as defined in the *AMBA Protocol Specifications* for missing signals.
- Performs lock transaction bit conversion when an AXI3 master connects to an AXI4 slave in 1x1 systems.

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 6-9: 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

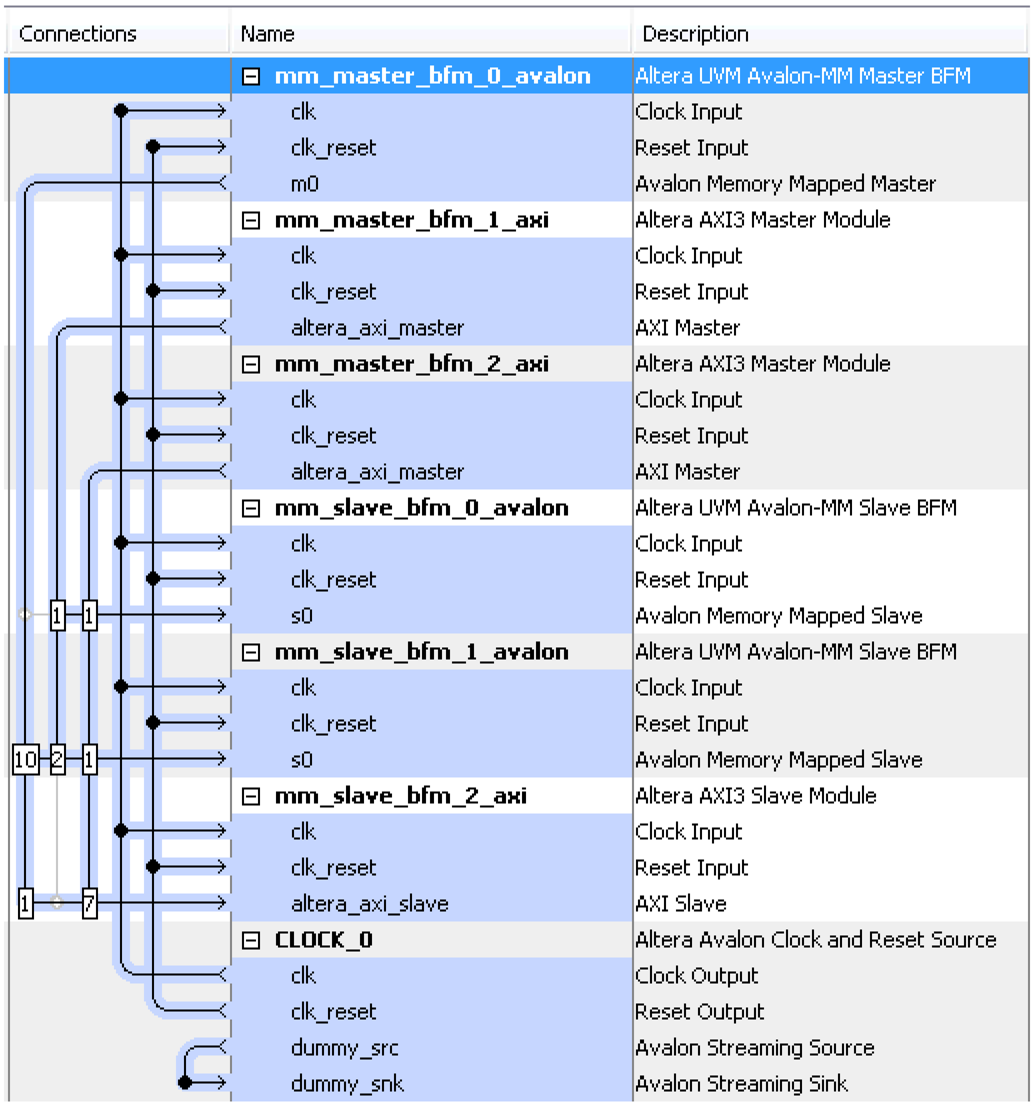
When multiple masters contend for access to a slave, Qsys automatically inserts arbitration logic, which grants access in fairness-based, round-robin order. You can alternatively choose to designate a slave as a fixed priority arbitration slave, and then manually assign priorities in the Qsys GUI.

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 6-10: 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 6-11: 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.

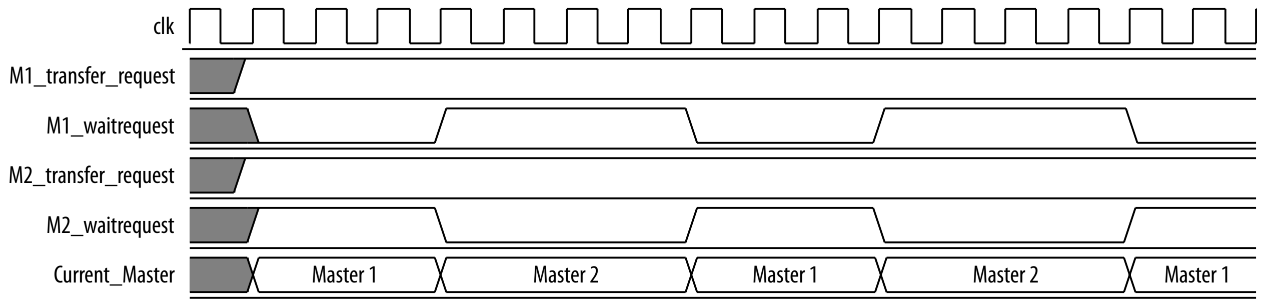
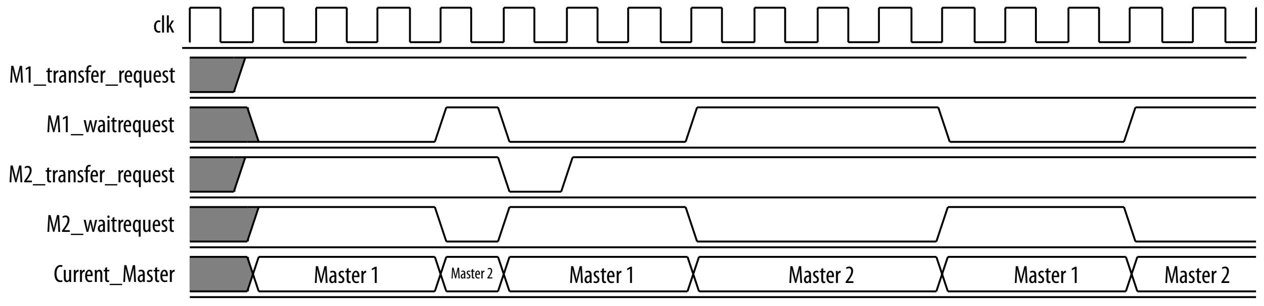


Figure 6-12: Arbitration of Two Masters with a Gap in Transfer Requests

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.

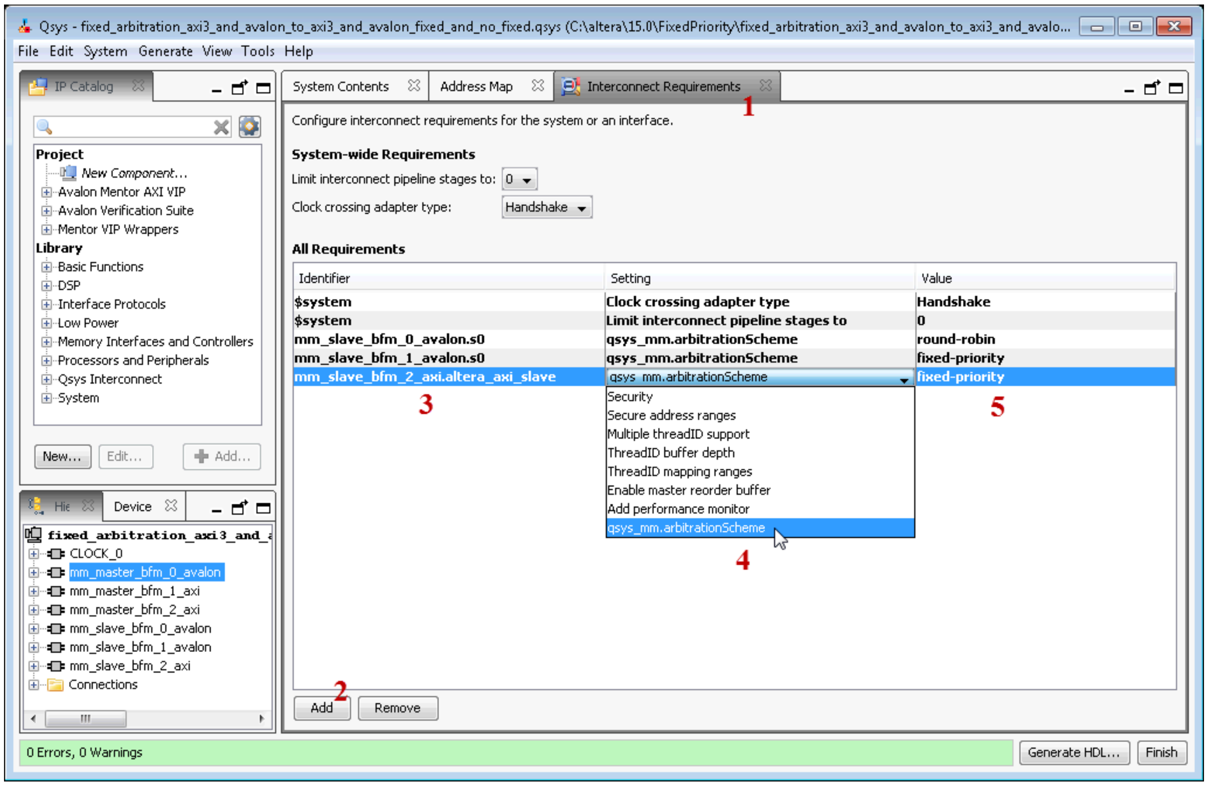
Note: When you connect an AXI master to an Avalon-MM slave designated to use a fixed priority arbitrator, the interconnect instantiates a command-path intermediary round-robin multiplexer in front of the designated slave.

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 6-13: 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

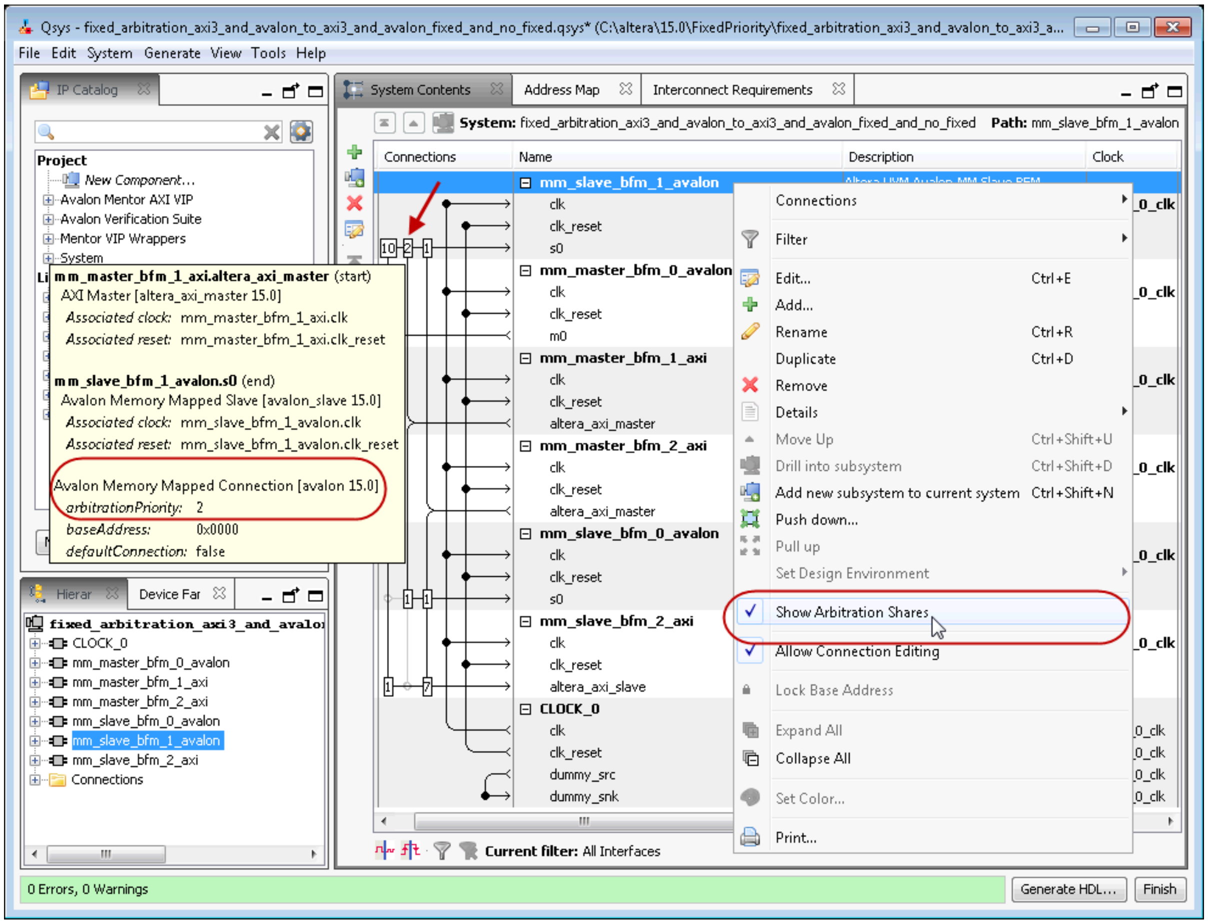
Identifier	Setting	Value
$system	Clock crossing adapter type	Handshake
$system	Limit interconnect pipeline stages to	0
mm_slave_bfm_0_avalon.s0	qsys_mm.arbitrationScheme	round-robin
mm_slave_bfm_1_avalon.s0	qsys_mm.arbitrationScheme	fixed-priority
mm_slave_bfm_2_axi.altera_axi_slave	qsys_mm.arbitrationScheme	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 6-14: 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

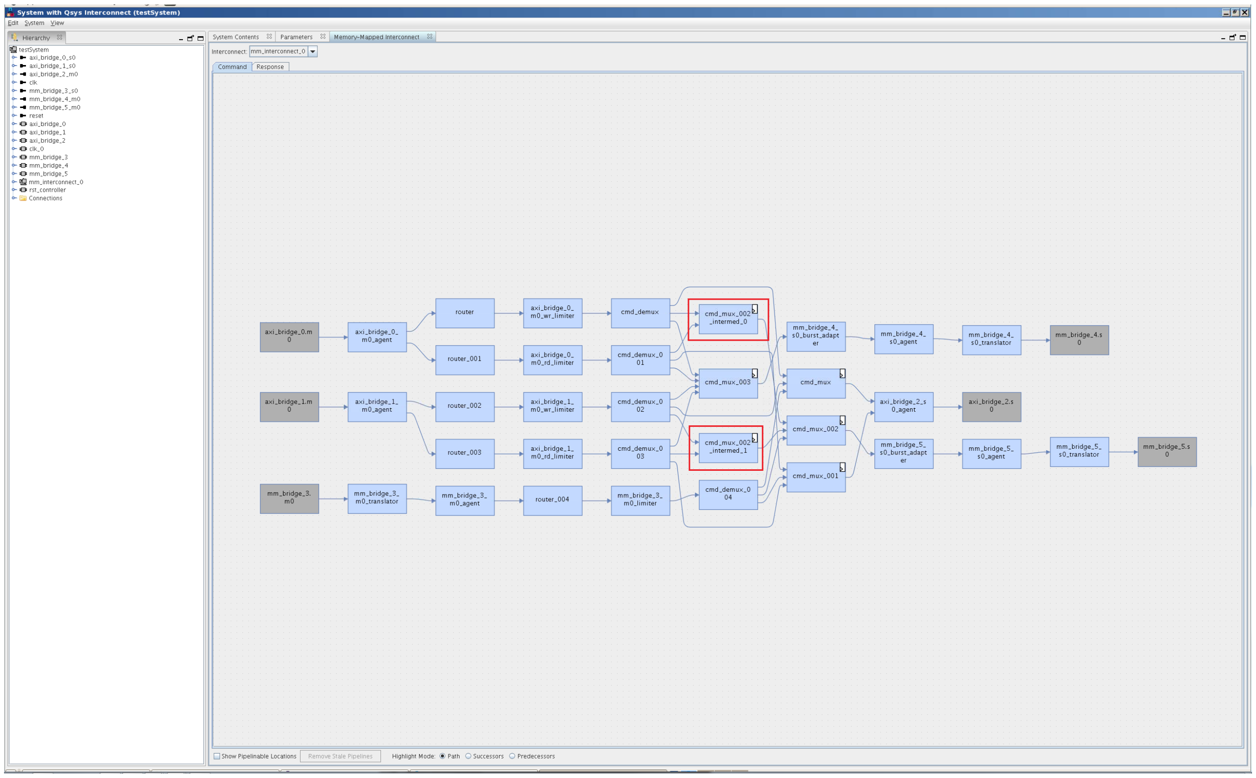
When an AXI master is connected to a designated fixed priority arbitration Avalon-MM slave, Qsys interconnect automatically instantiates an intermediary multiplexer in front of the Avalon-MM slave.

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 6-15: Intermediary Multiplexer Between AXI Master and Avalon-MM Slave

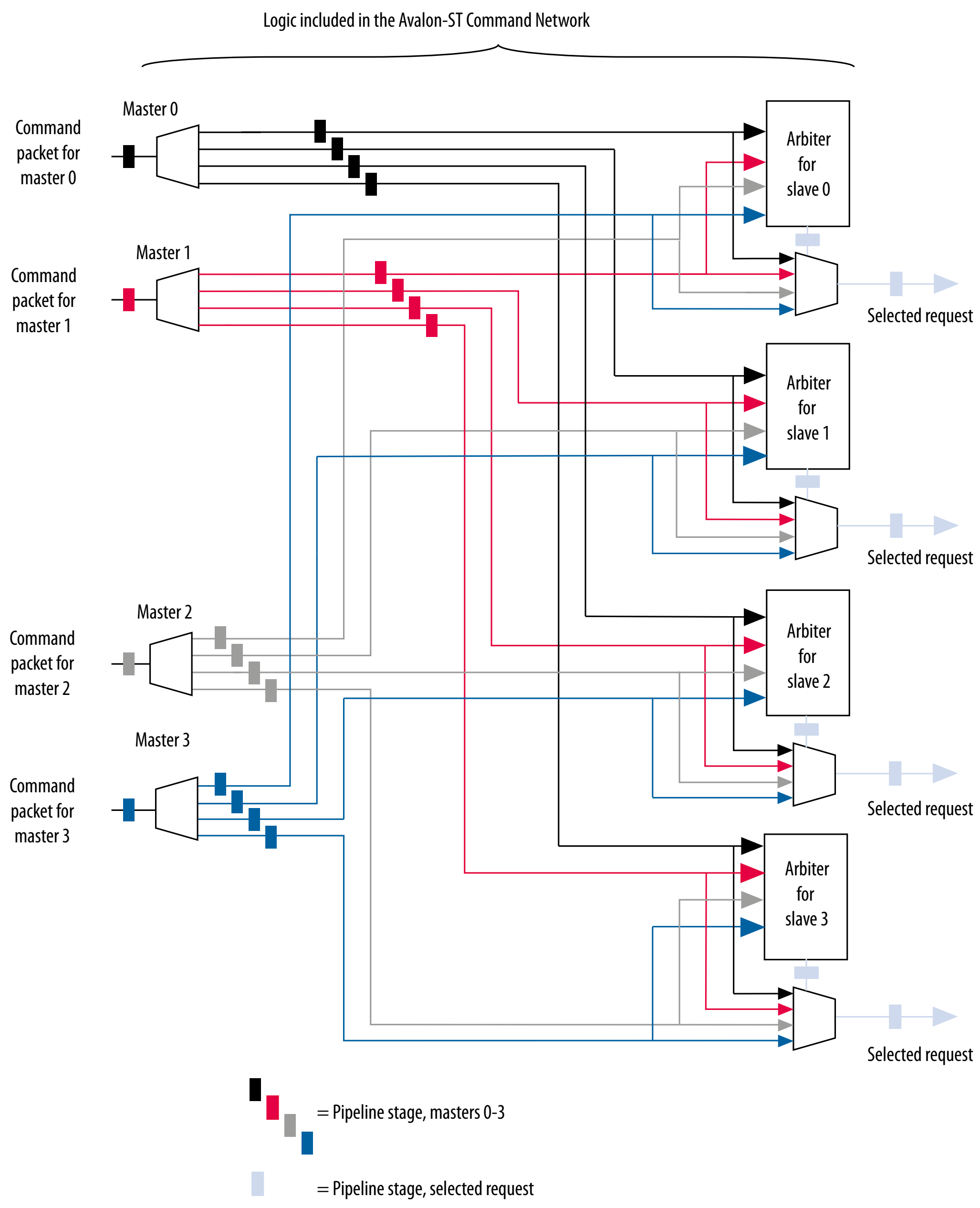


Memory-Mapped Arbiter

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 6-16: 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

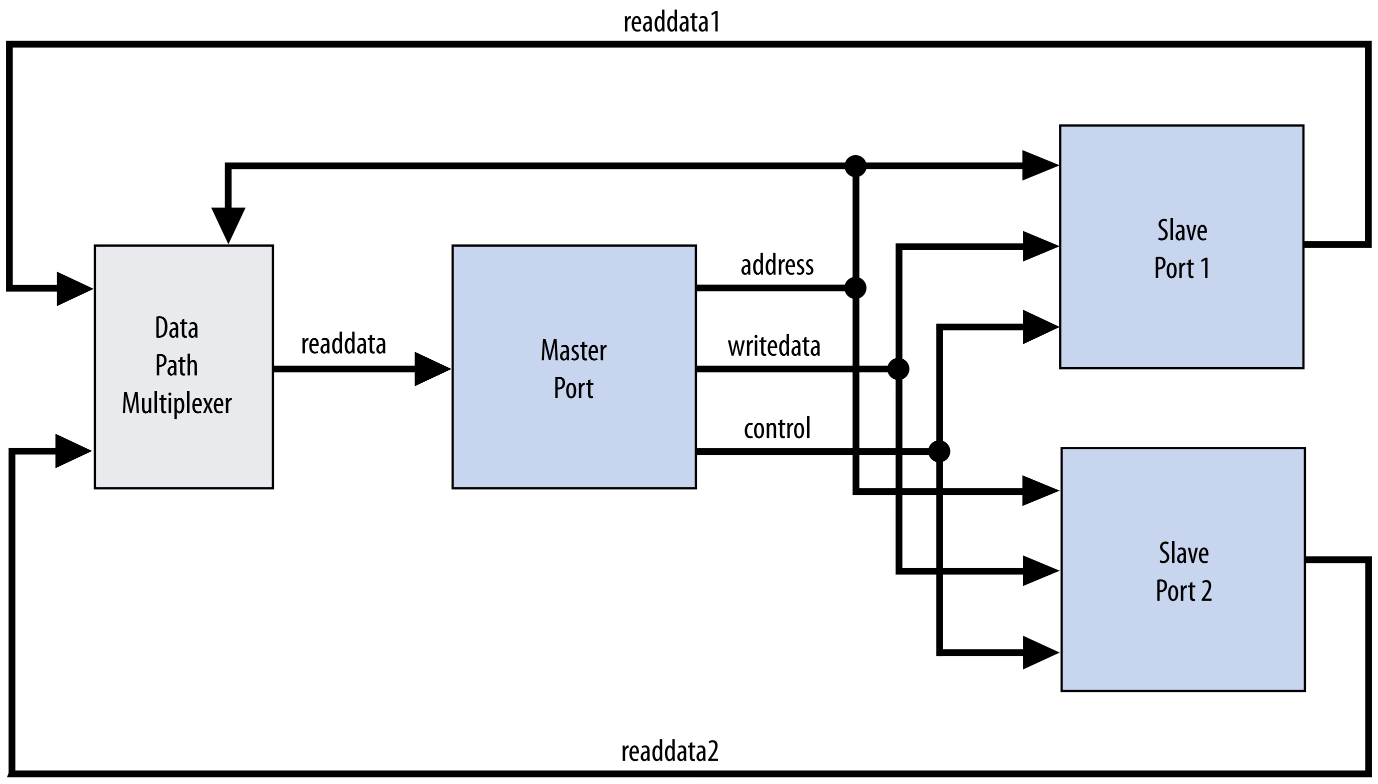
Note: If you specify a **Limit interconnect pipeline stages to** parameter greater than zero, the output of the Arbiter is registered. Registering this output reduces the amount of combinational logic between the master and the interconnect, increasing the f_{MAX} of the system.

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 6-17: 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

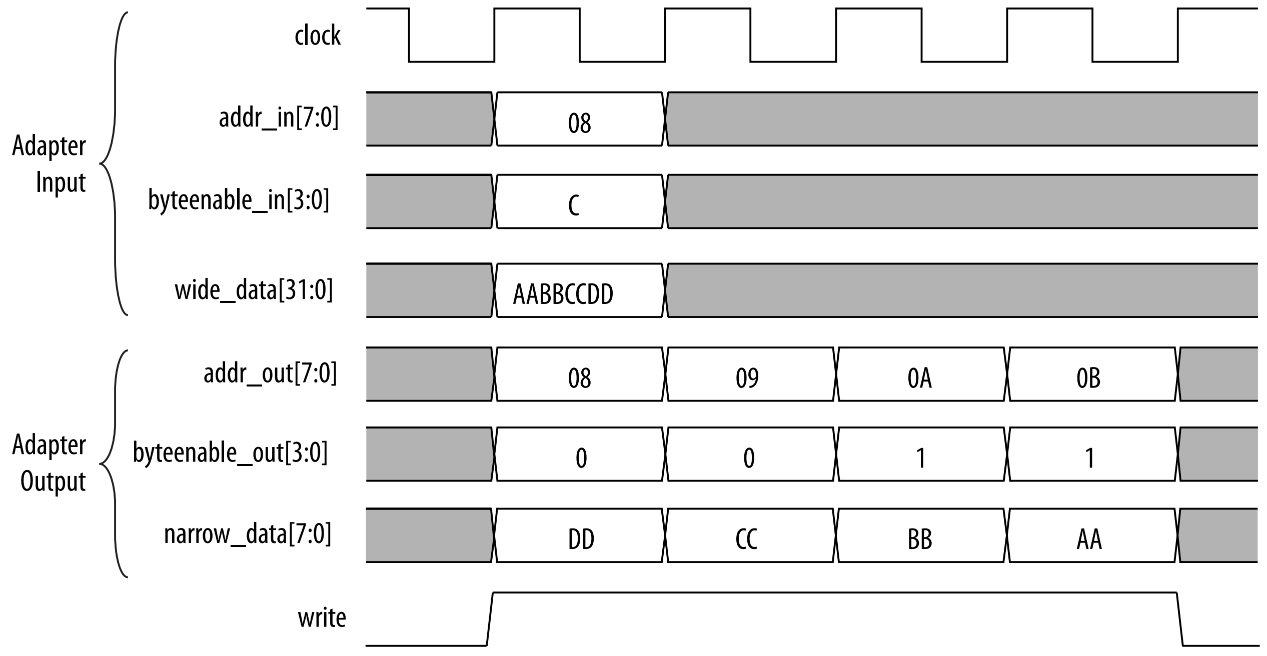
The Memory-Mapped Width Adapter is used in the Avalon-ST domain and operates with information contained in the packet format.

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 6-18: 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 6-3: 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 6-4: 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 4-1

Burst Adaptation: AXI to Avalon

Table 6-5: 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 6-6: 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.

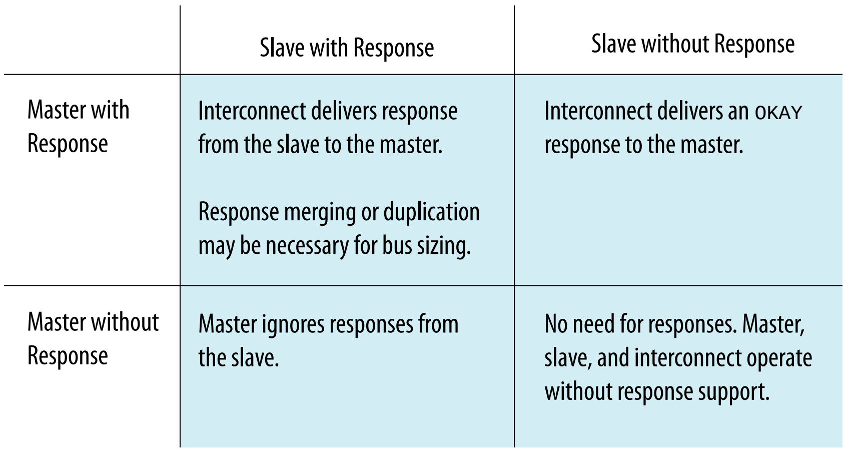
Qsys merges responses based on the following precedence rule:

```
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 6-19: 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.

Note: If there is a bridge between the master and the endpoint slave, and the responses must come from the endpoint slave, ensure that the bridge passes the appropriate response signals through from the endpoint slave to the master.

If the bridge does not support responses, then the responses are generated by the interconnect at the slave interface of the bridge, and responses from the endpoint slave are ignored.

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 6-11
- **Error Correction Coding (ECC) in Qsys Interconnect** on page 6-69
- **Avalon-MM Interface Signal Roles**
- **Interface Properties**

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 6-20: 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.

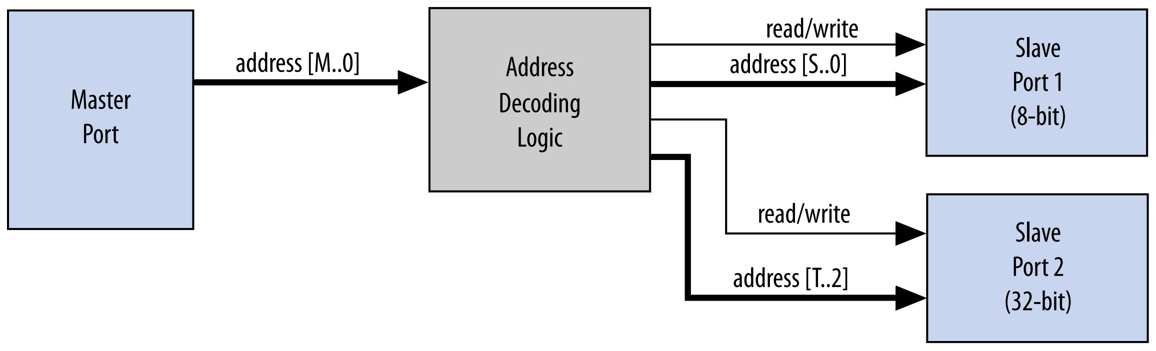
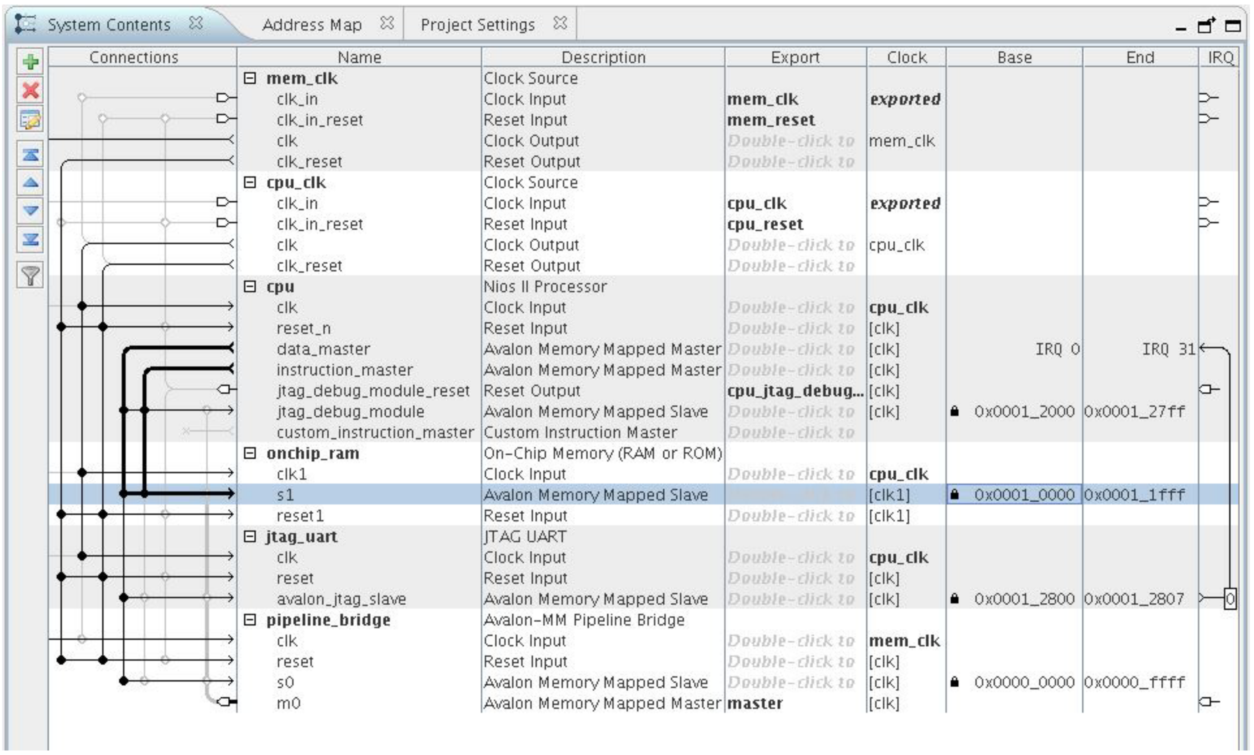


Figure 6-21: Address Decoding Base Settings

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}.



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 6-22: 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**.

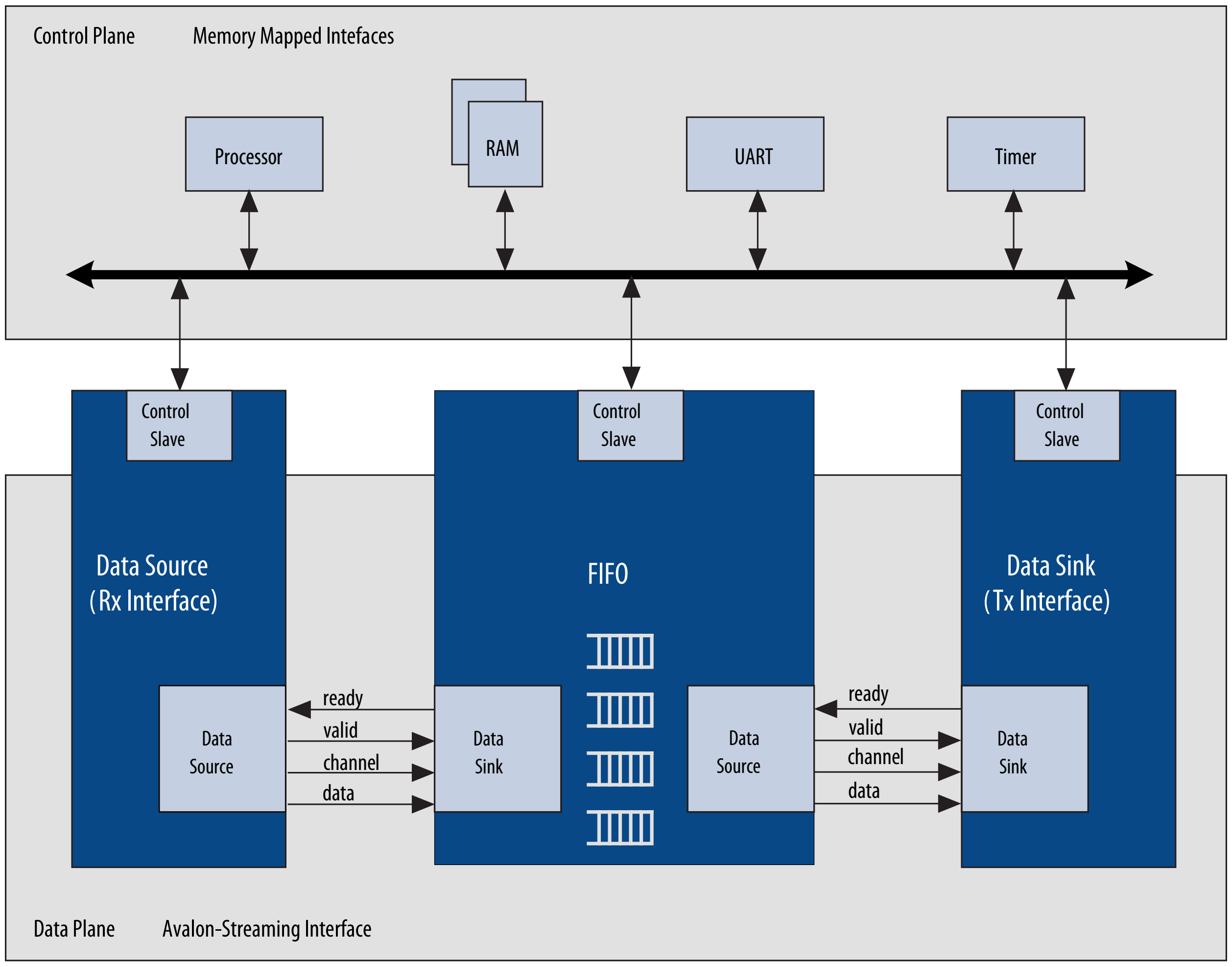


Figure 6-23: Avalon-ST Connection Between the Source and Sink

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.

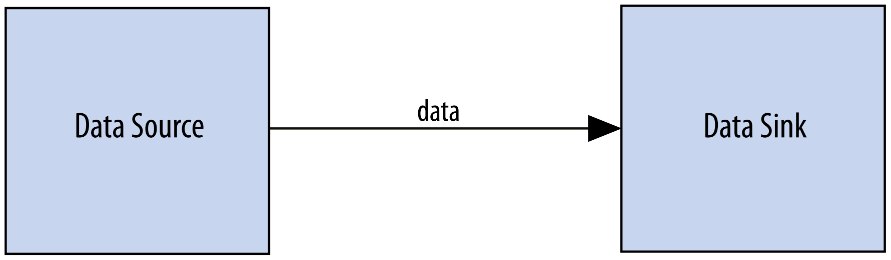
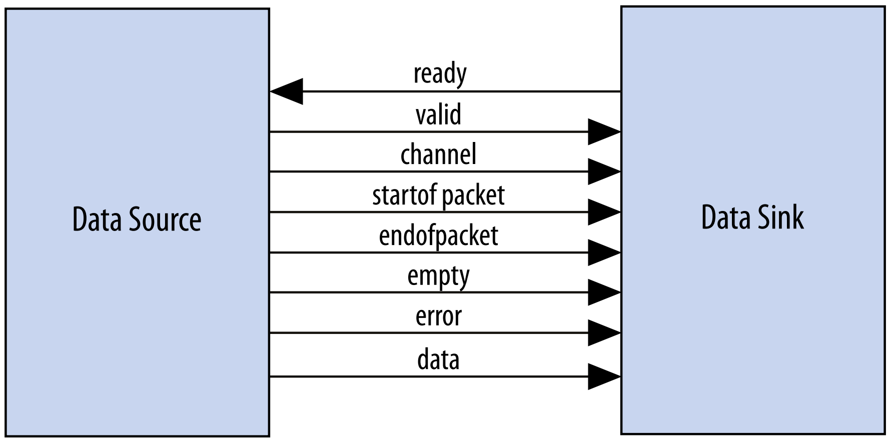


Figure 6-24: Signals Indicating the Start and End of Packets, Channel Numbers, Error Conditions, and Backpressure

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 6-31
- **Qsys Transformations** on page 6-7
- **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 6-7: 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 6-8: 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 6-9: 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 6-10: 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 6-11: 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 6-12: 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 6-13: 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

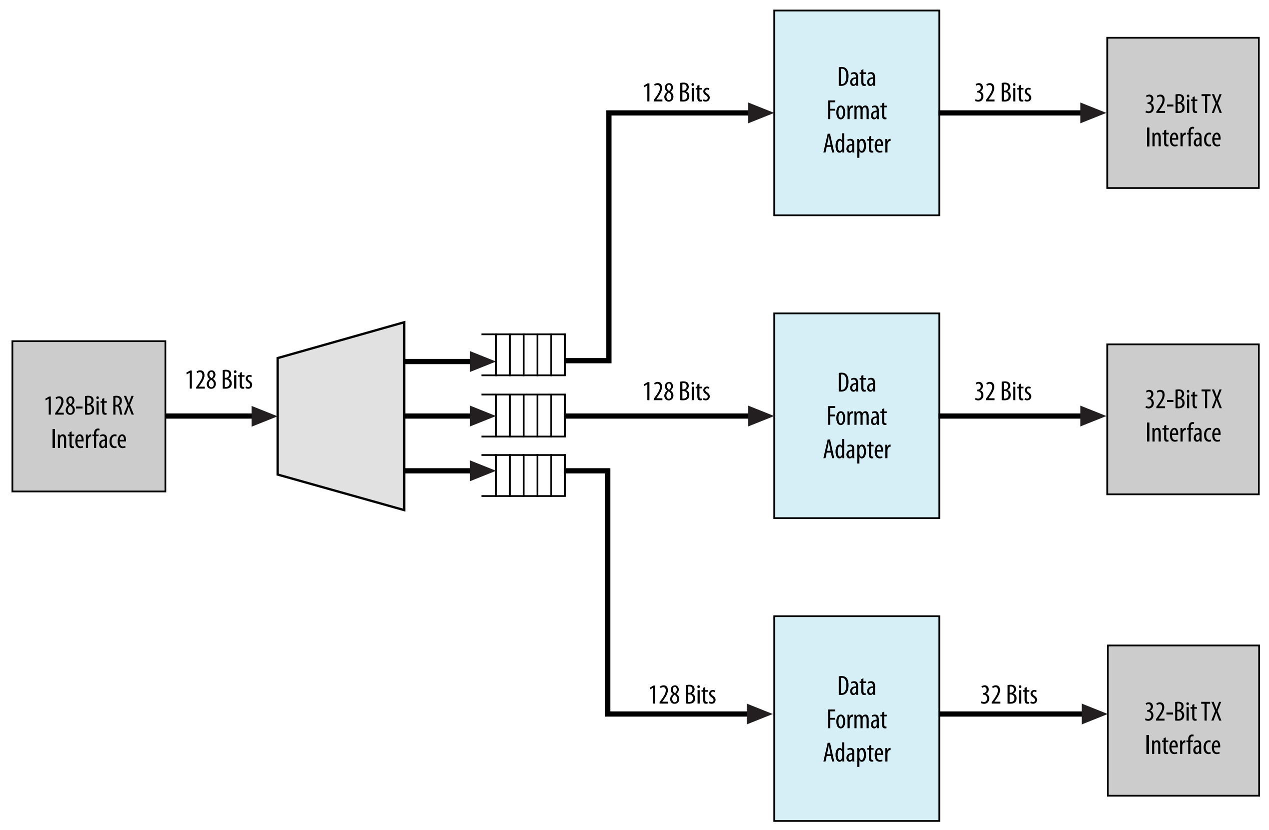
The data format adapter allows you to connect interfaces that have different values for the parameters defining the `data` signal, or interfaces where the source does not use the `empty` signal, but the sink does use the `empty` signal. One of the most common uses of this adapter is to convert data streams of different widths.

Table 6-14: 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 6-25: 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 6-15: 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 6-16: 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 6-17: 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 6-18: 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 6-19: 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 6-20: 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 function-ality 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 6-21: Timing Adapter Adaptations

Condition	Adaptation
The source has `ready`, but the sink does not.	In this case, the source can respond to `backpres-sure`, 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 deasser-tion 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 6-22: 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 6-23: 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 6-24: 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 `startof-packet`, `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 6-44

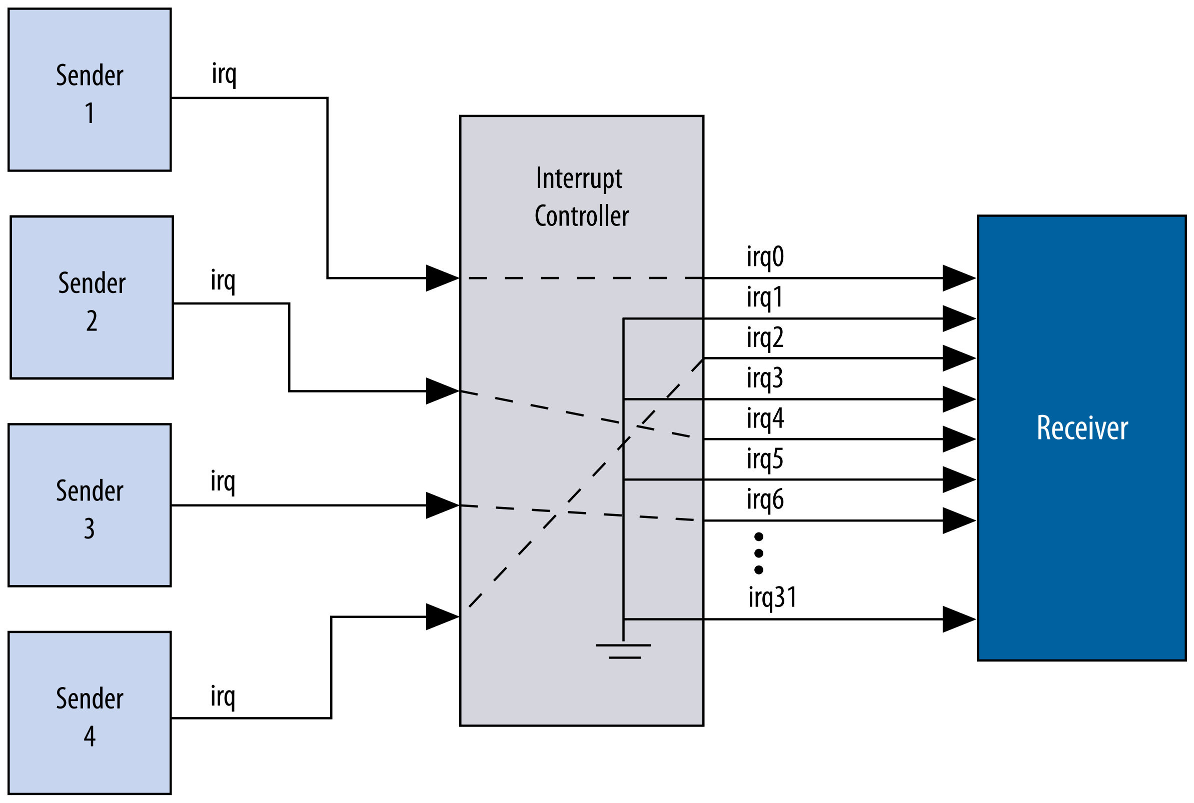
Individual Requests IRQ Scheme

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 6-26: 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

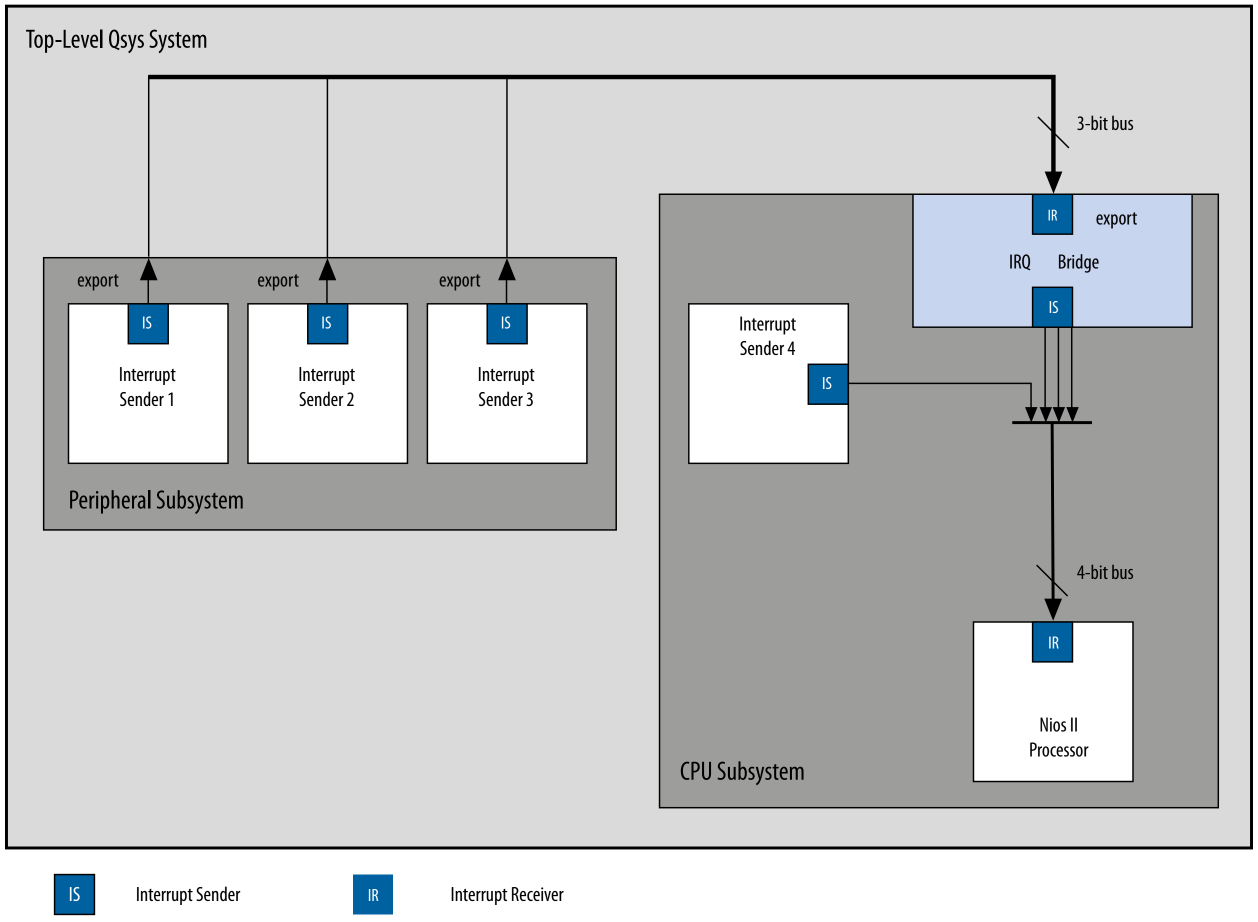
You assign IRQ connections on the **System Contents** tab of Qsys. After adding all components to the system, you connect interrupt senders and receivers. You can use the **IRQ** column to specify an IRQ number with respect to each receiver, or to specify a receiver's IRQ as unconnected. Qsys uses the 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 6-27: 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.

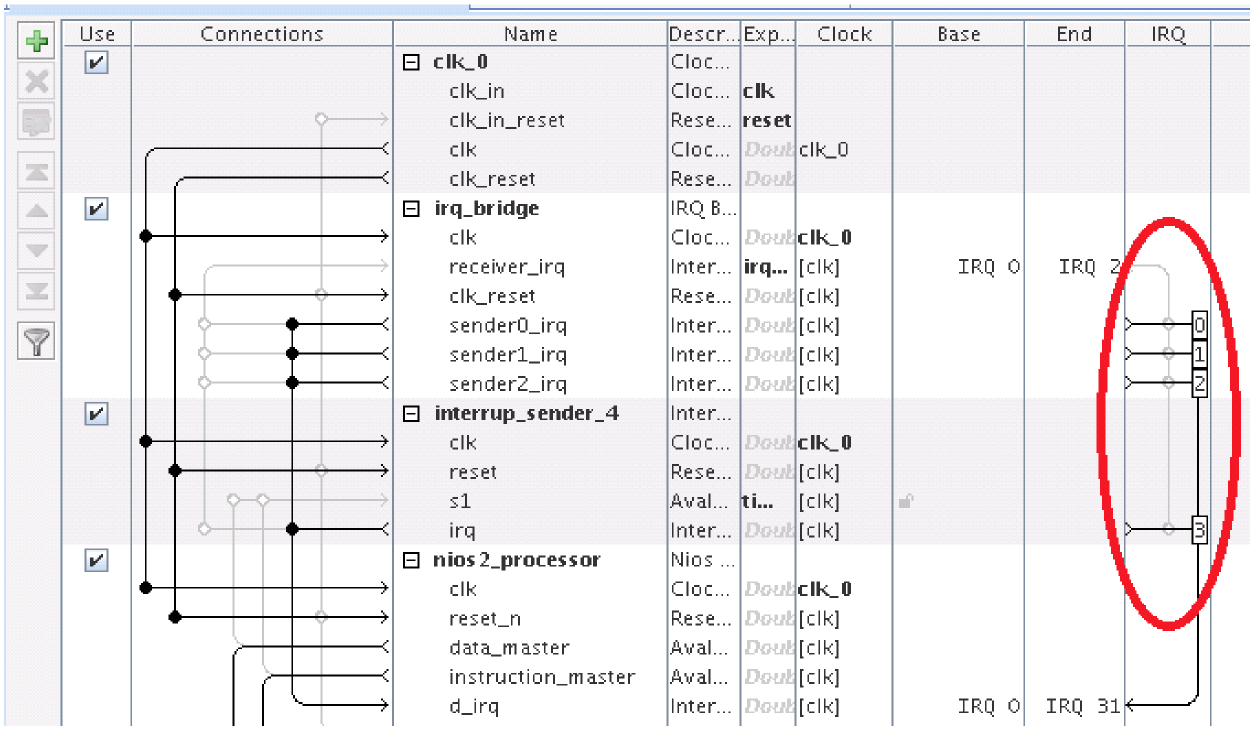
IRQ Mapper

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 6-28: 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 6-43

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 10-1

(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 6-25: 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 6-26: 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 6-27: 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 6-28: 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 6-1: 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 6-2: 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 6-29: 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 6-30: 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 6-31: 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 6-32: 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 6-3: 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 6-4: 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:

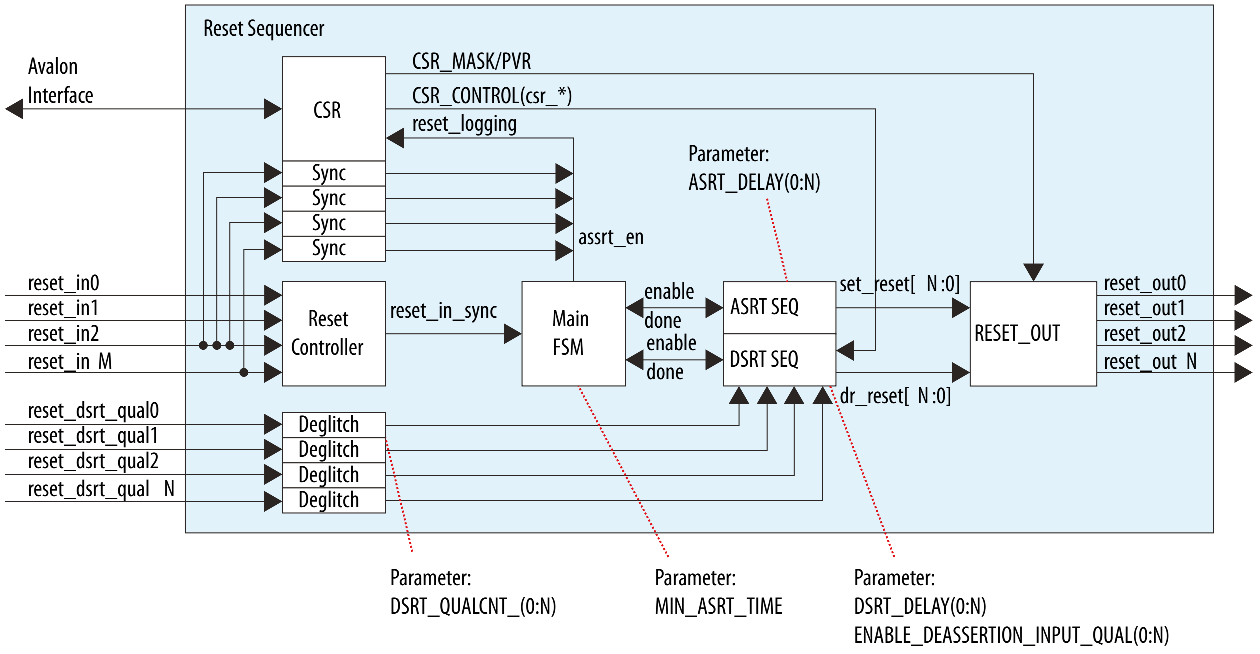
- **Active low reset**—When turned on, reset is asserted low.
- **Synchronous edges**—Specifies the level of synchronization and includes the following options:
  - **None**—The reset is asserted and deasserted asynchronously. Use this setting if you have internal synchronization circuitry.
  - **Both**—The 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 6-29: 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 6-33: 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 6-30: Basic Sequencing

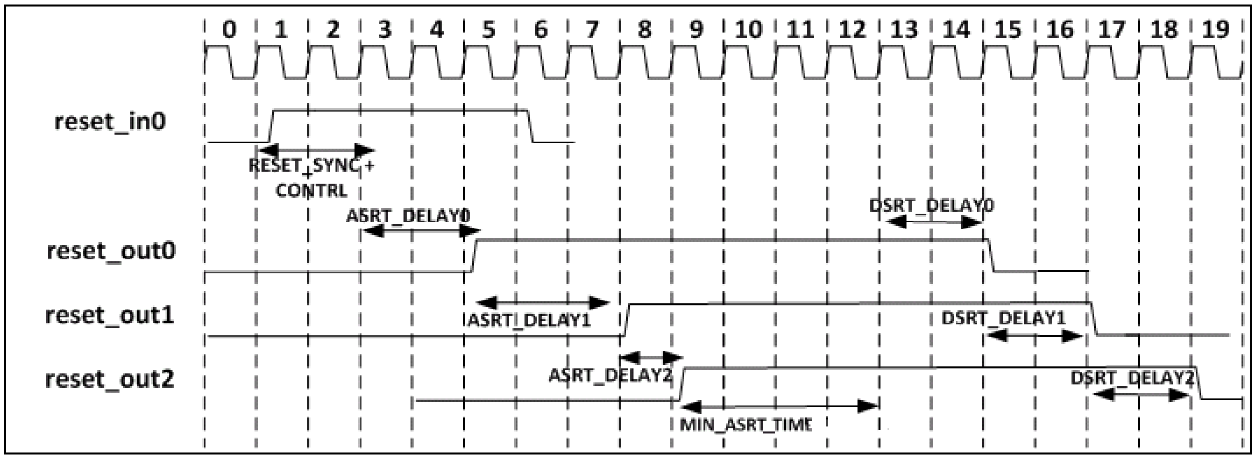
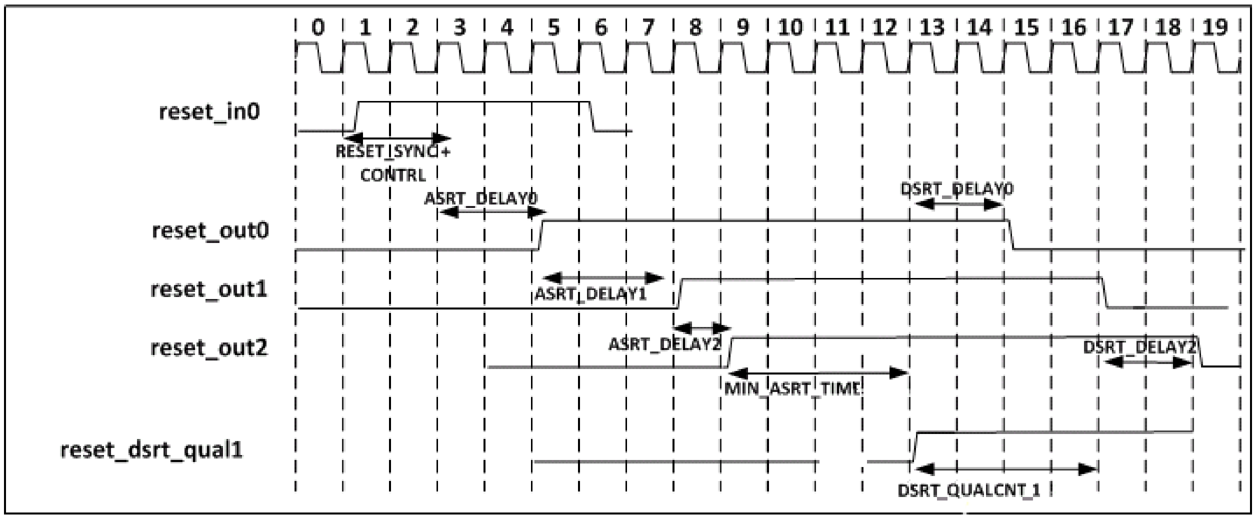


Figure 6-31: Sequencing with USE_DSRT_QUAL Set



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 6-34: 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 6-35: 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 6-36: 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 6-37: 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 6-38: 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 6-39: 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.

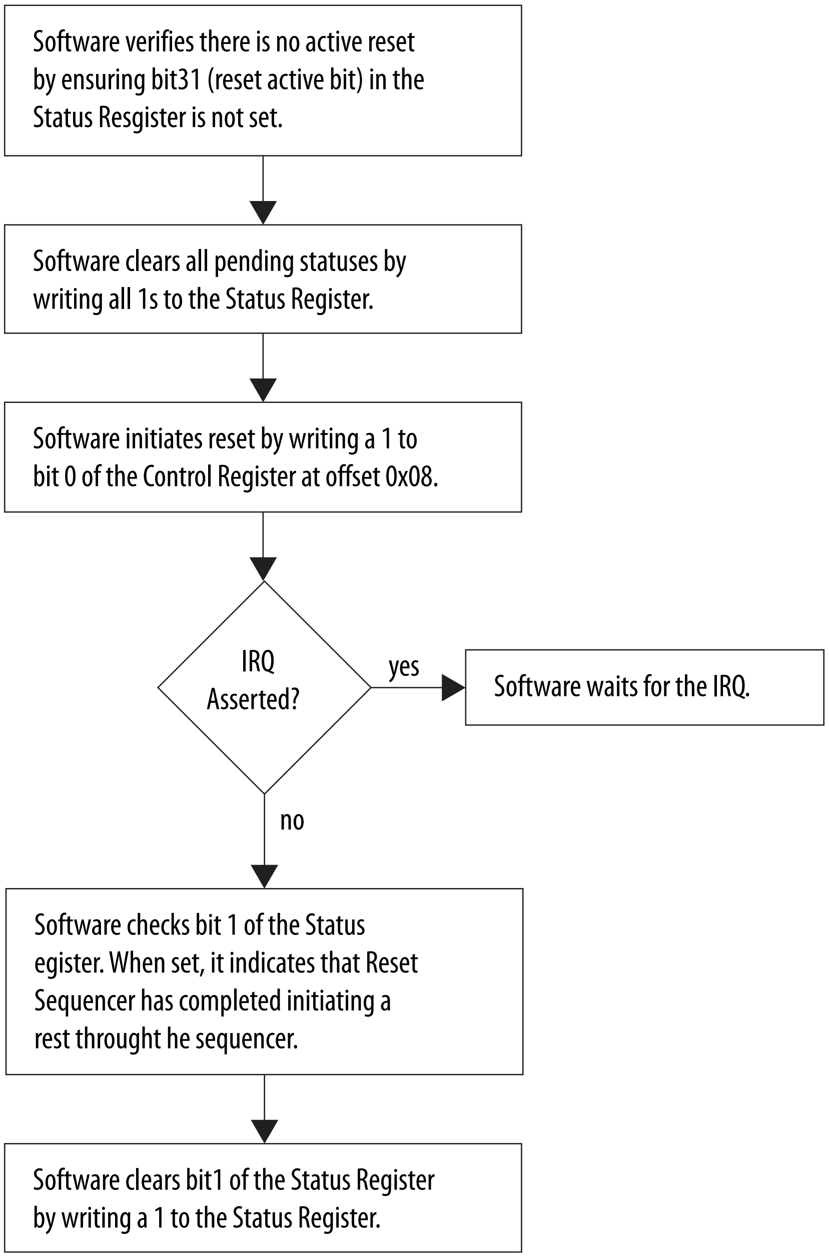
Table 6-40: Values for the Reset Sequencer Software Reset Masking at Offset 0x18

Bit	Attrib ute	Defaul t	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 6-32: 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.

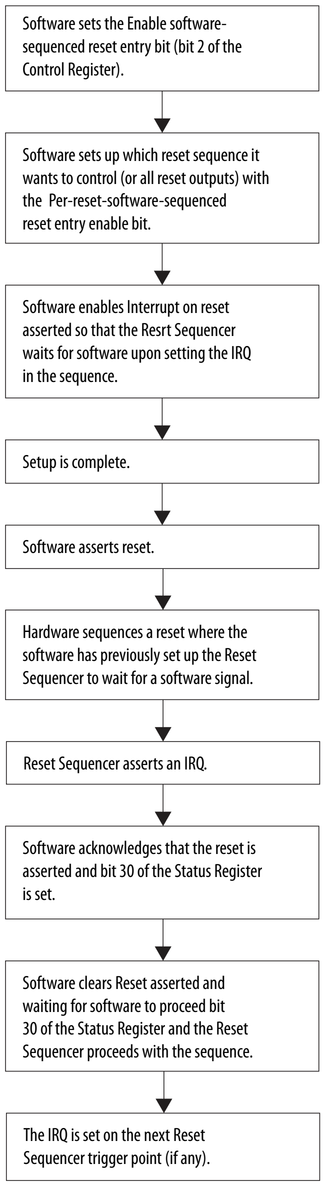
Reset Bring-Up Flow

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 6-33: 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 6-65

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 4-1

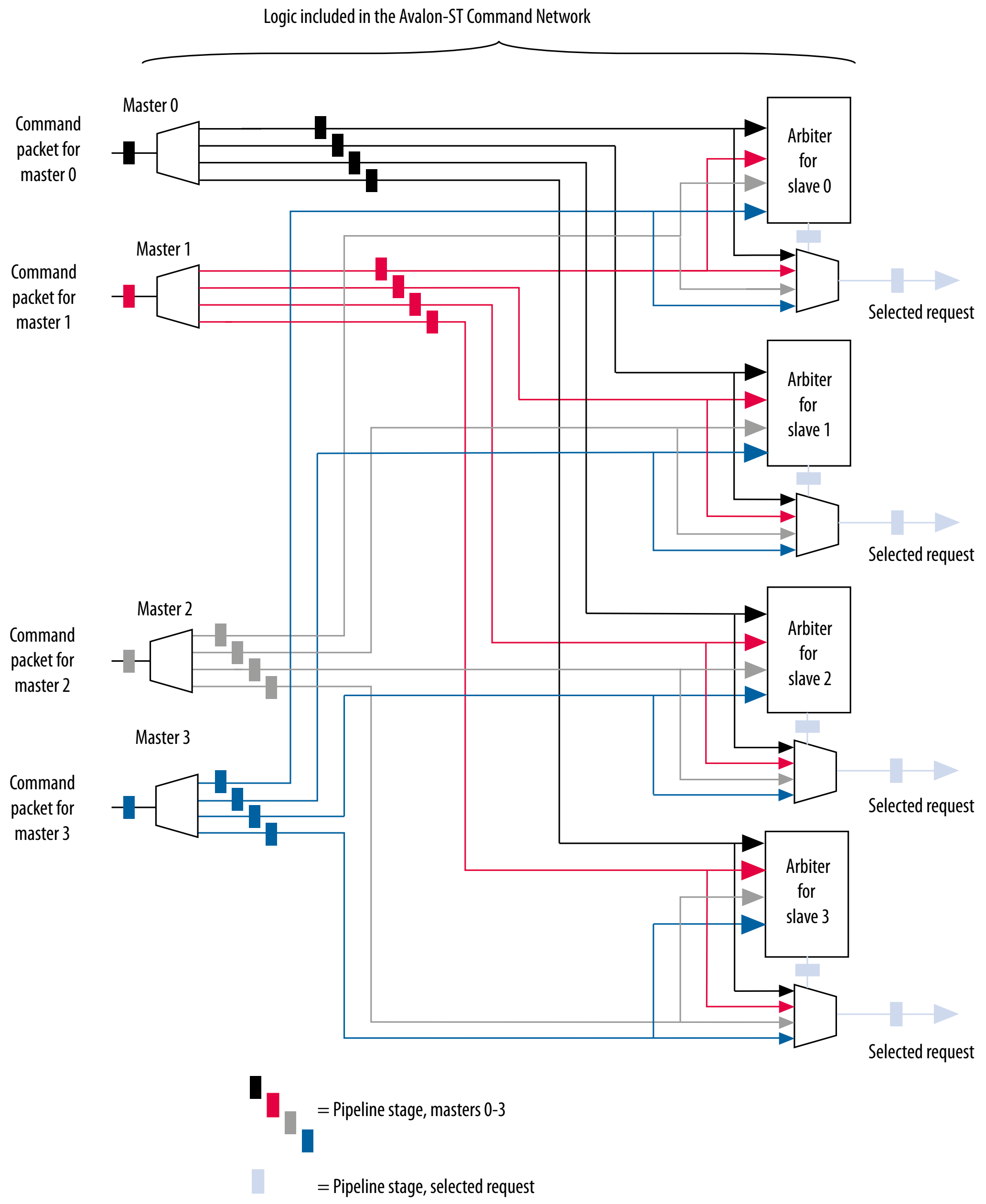
Interconnect Pipelining

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 6-34: 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 4-1

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 Prime 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 9-1

Error Correction Coding (ECC) in Qsys Interconnect

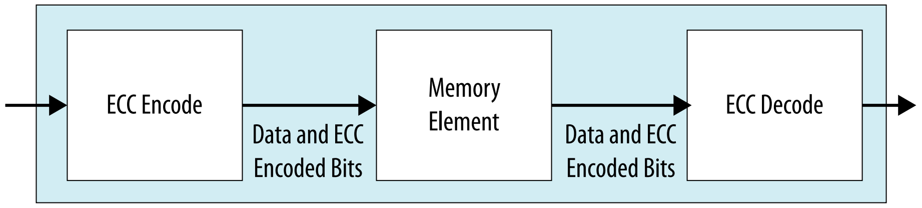
Error Correction Coding (ECC) allows the Qsys interconnect to detect and correct errors in order to improve data integrity in memory blocks.

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 6-35: High-Level Implementation of RDATA FIFO with ECC Enabled



Related Information

- **Read and Write Responses** on page 6-27
- **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 4-1

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 6-41: 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 6-42: AXI4 Stream Connection Point Signals

Port Role	Width	Master Direction	Slave Direction	Required
`tvalid`	1	Output	Input	Yes
`tready`	1	Input	Output	No
`tdata`[4]	8:4096	Output	Input	No
`tstrb`	1:512	Output	Input	No
`tkeep`	1:512	Output	Input	No
`tid`[5]	1:8	Output	Input	No
`tdest`[6]	1:4	Output	Input	No
`tuser`[7]	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.

[4] integer in mutiple of bytes
[5] maximum 8-bits
[6] maximum 4-bits
[7] number of bits in multiple of the number of bytes of `tdata`

Table 6-43: 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 6-44: 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 6-45: 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 6-46: 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 6-47: 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 6-48: 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 6-49: 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 6-50: 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` 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. **Note:** The AXI interface supports unaligned accesses while Avalon-MM does not. Unaligned accesses going from an AXI master to an Avalon-MM slave may result in an illegal transaction.To avoid this issue, only use aligned accesses to Avalon-MM slaves.
`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 an 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: • One response is sent 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. • One write response must be sent for each write command. A write burst results in only one response, which must be sent 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 `beginburst-transfer` 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 6-51: 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 and 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 6-52: 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 6-53: 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 6-54: 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 6-55: 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 6-56: 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 6-57: 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 6-58: 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 6-59: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
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 Handbook Archive
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Optimizing Qsys System Performance 7

2015.11.02

QPP5V1 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 4-1
- Creating Qsys Components on page 5-1
- Qsys Interconnect on page 6-1

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 5-1

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.

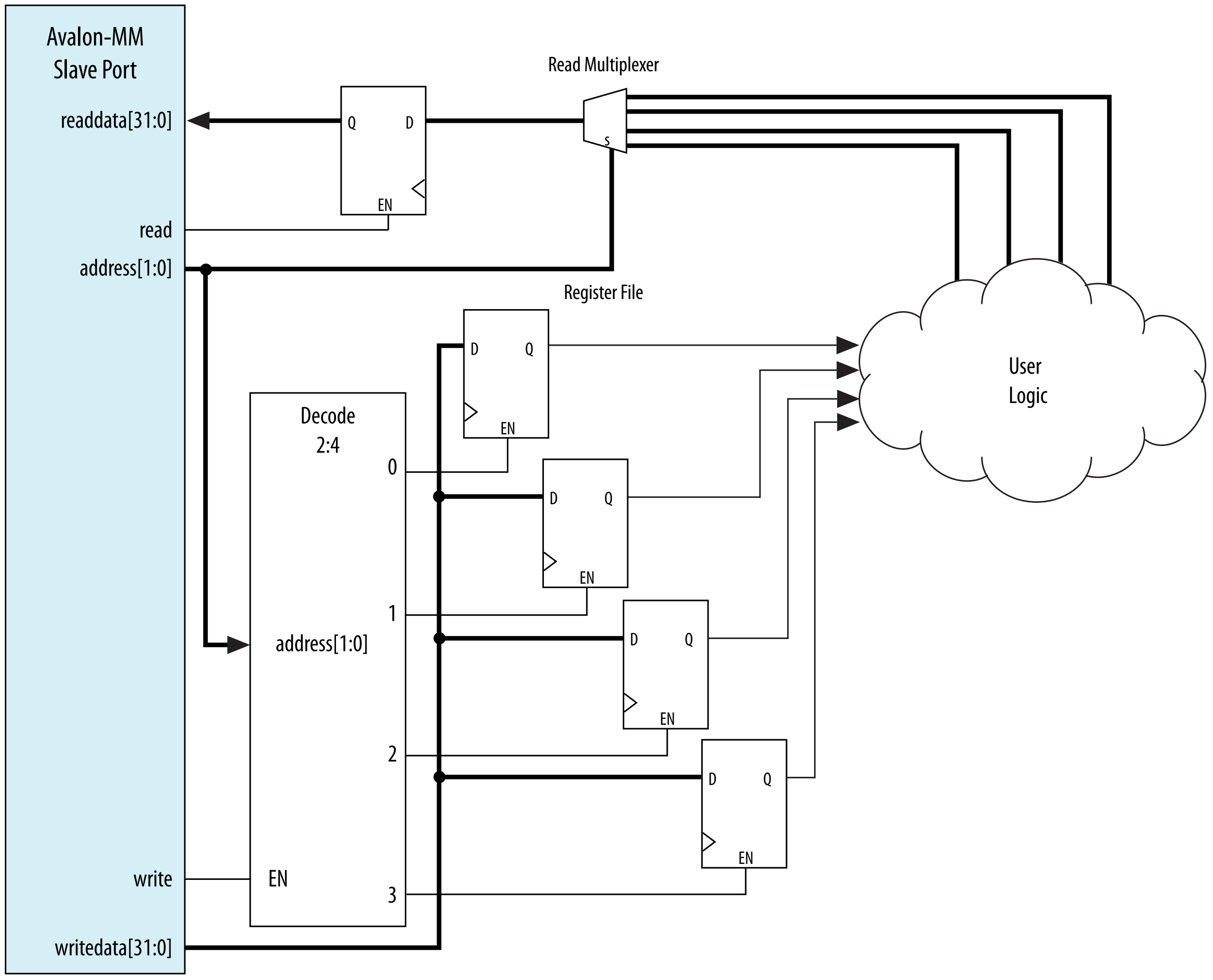
You can use a FIFO to back-pressure internally on the output side of the component so that the input can accept more data even if the output is back-pressured. Then, you can use the FIFO almost full flag to back-pressure the sink interface or input data when the FIFO has only enough space to satisfy 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 7-1: Control and Status Registers (CSR) in a Slave Component



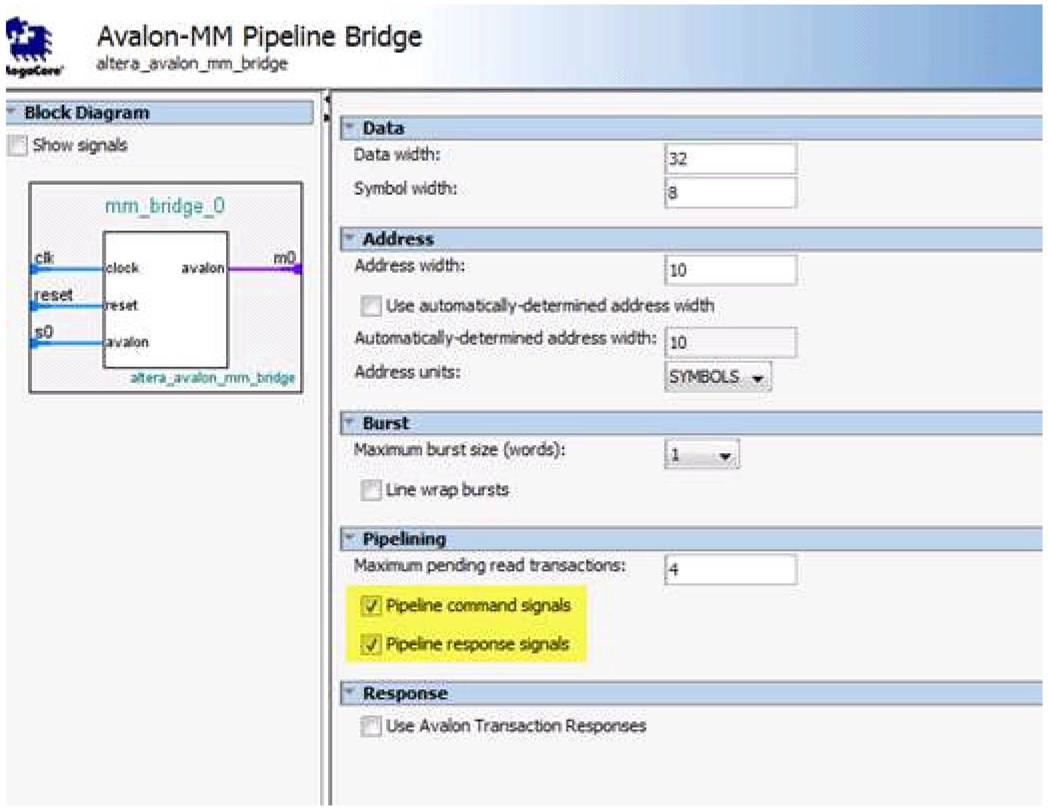
This slave component has write wait states and one read wait state. Alternatively, if you want high throughput, you may set both the read and write wait states to zero, and then specify a read latency of one, because the component also supports pipelined reads.

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 an Avalone-MM Pipeline Bridge between subsystems, you may add latency to the overall system. You can reduce the added latency by parameterizing the bridge with zero cycles of latency, and by turning off the pipeline command and response signals.



Data
Data width: 32
Symbol width: 8

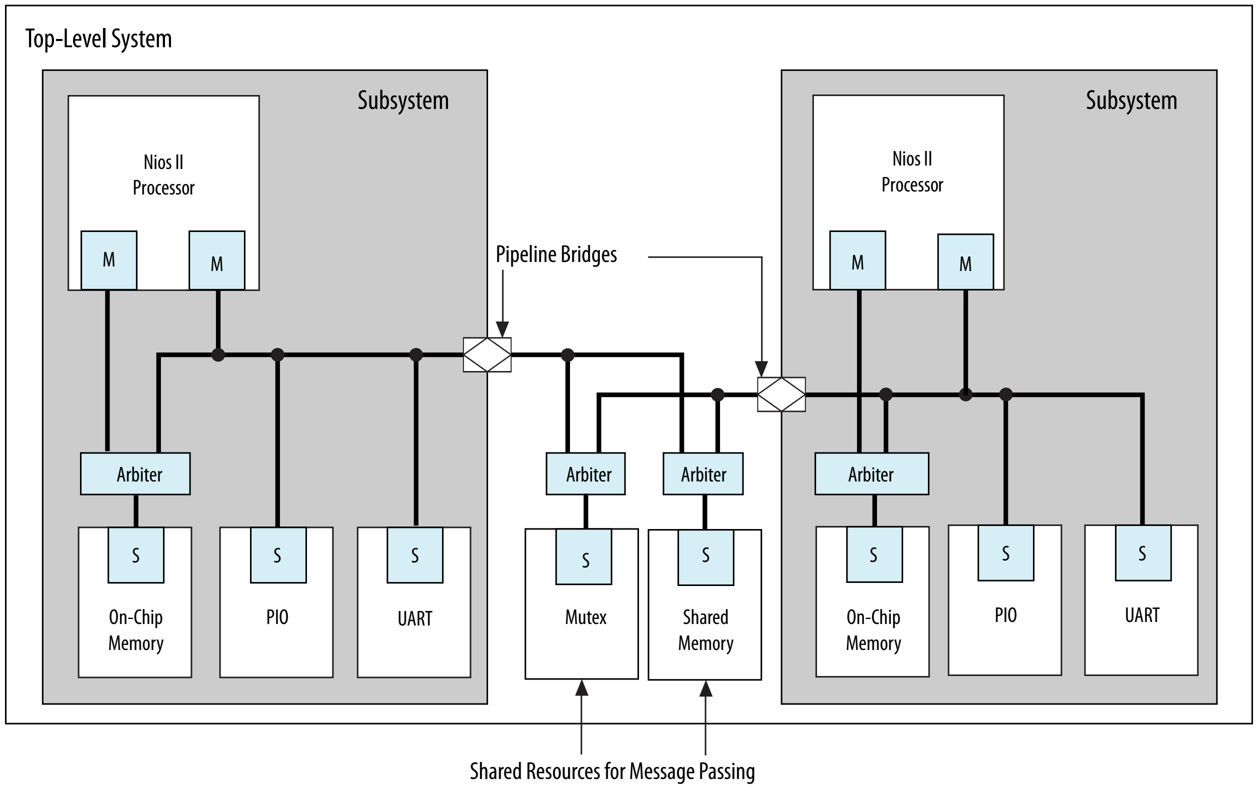
Address
Address width: 10
[ ] Use automatically-determined address width
Automatically-determined address width: 10
Address units: SYMBOLS

Burst
Maximum burst size (words): 1
[ ] Line wrap bursts

Pipelining
Maximum pending read transactions: 4
[✓] Pipeline command signals
[✓] Pipeline response signals

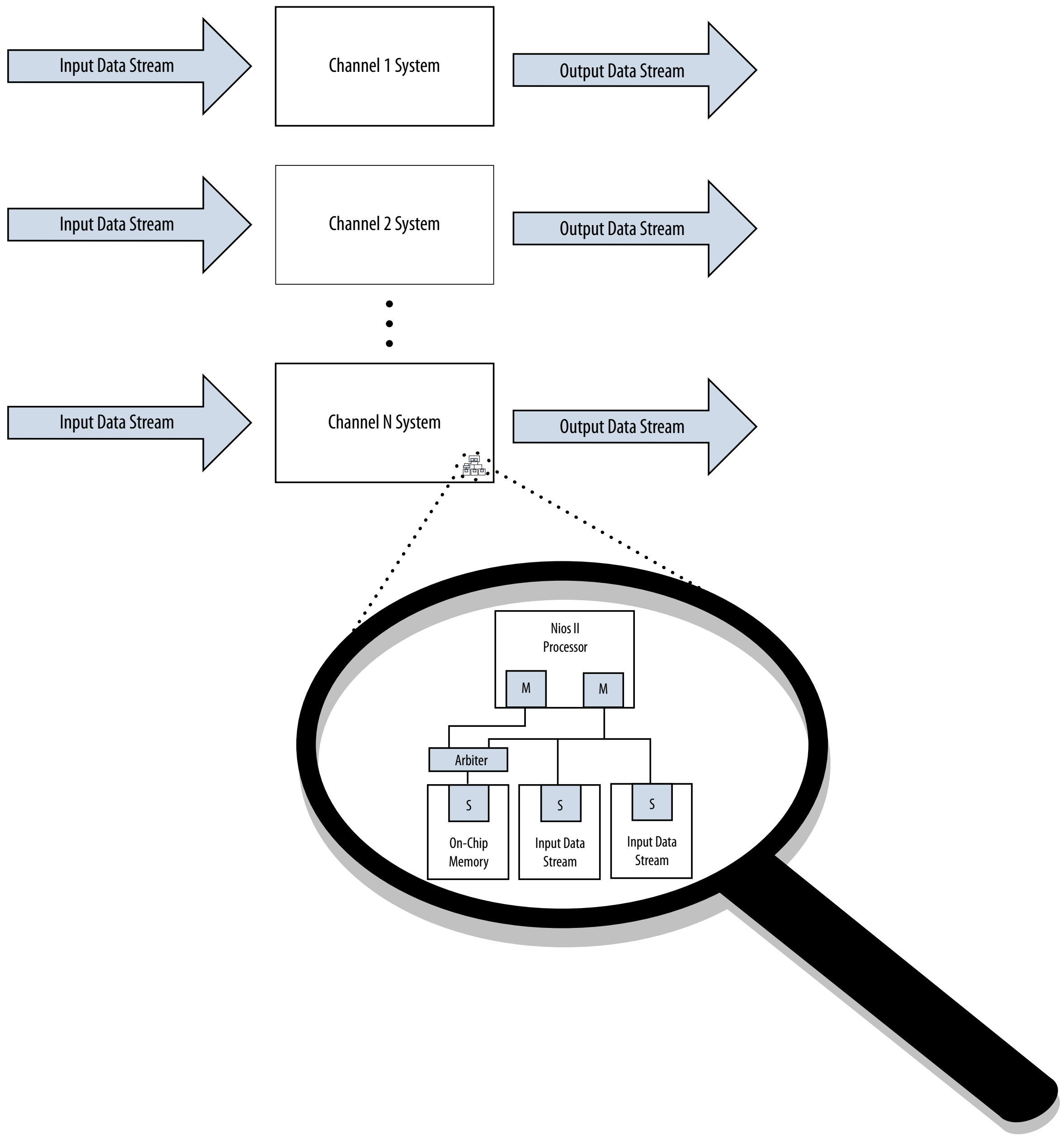
Response
[ ] Use Avalon Transaction Responses

Figure 7-2: 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 7-3: 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 mwh1409959480842 because you can mwh1409959275749 calculate the required resources as a multiple of the subsystem requirements.

Related Information

Avalon-MM Pipeline Bridge

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 7-4: 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.

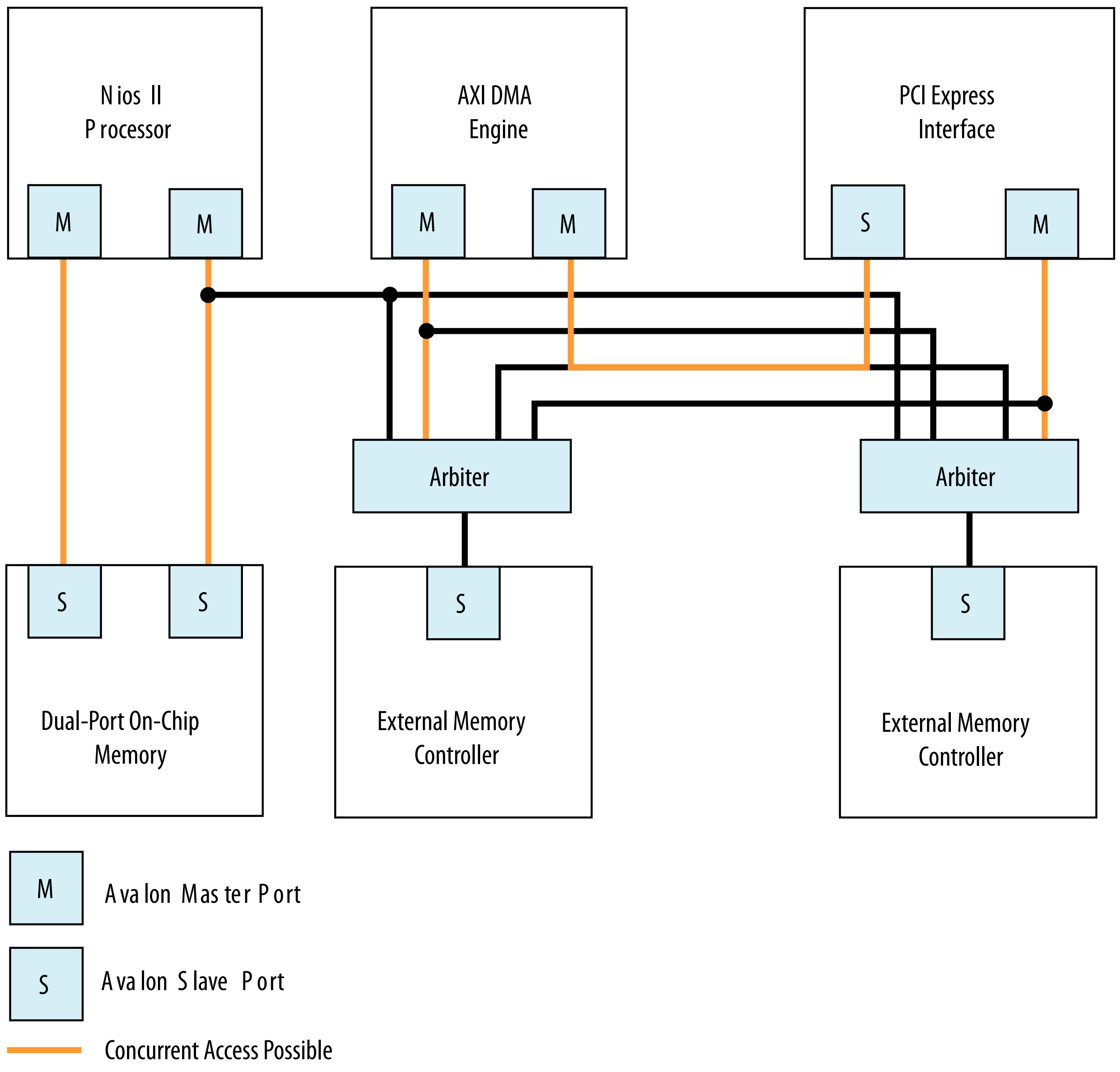
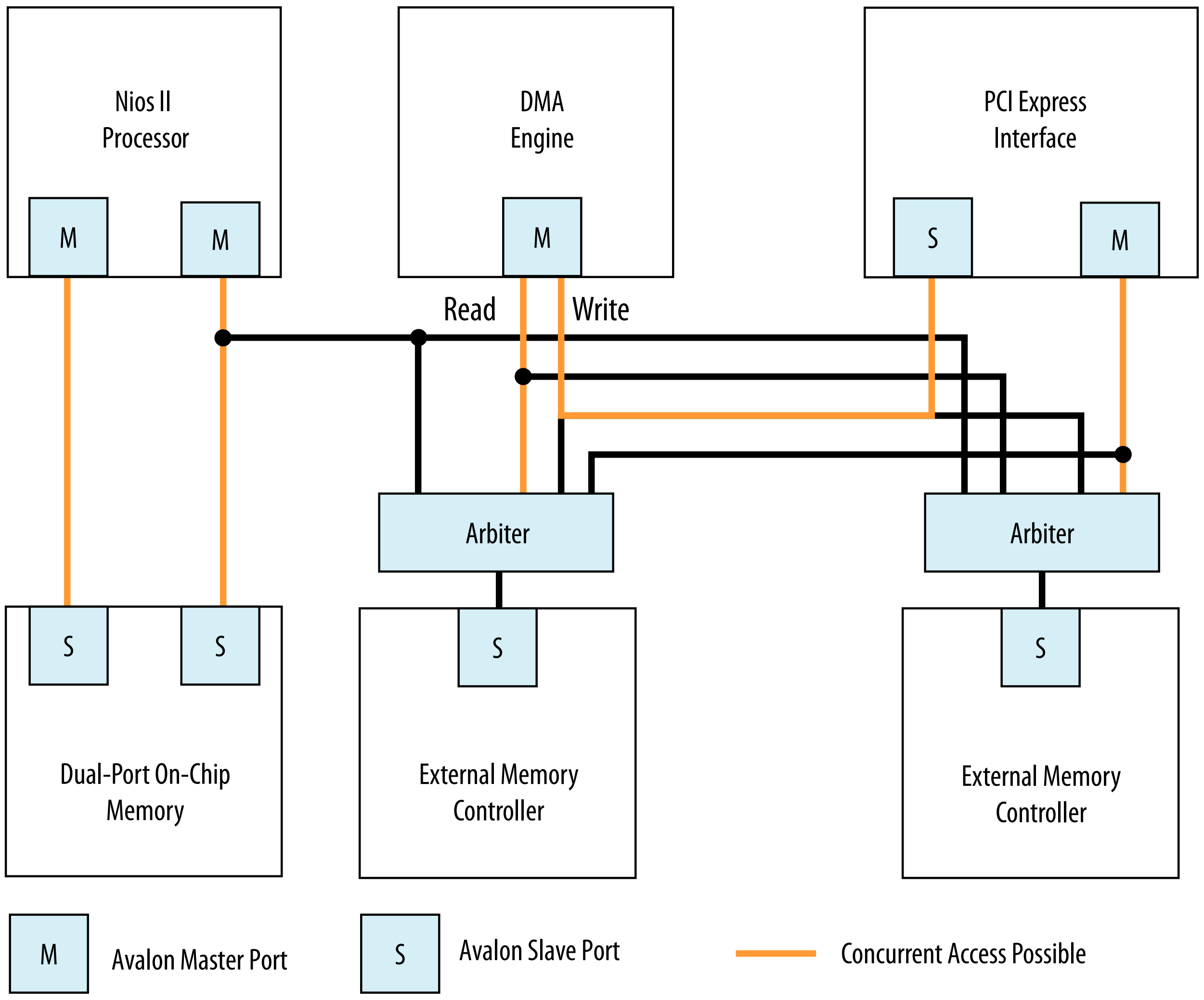


Figure 7-5: AXI Multiple Master Parallel Access

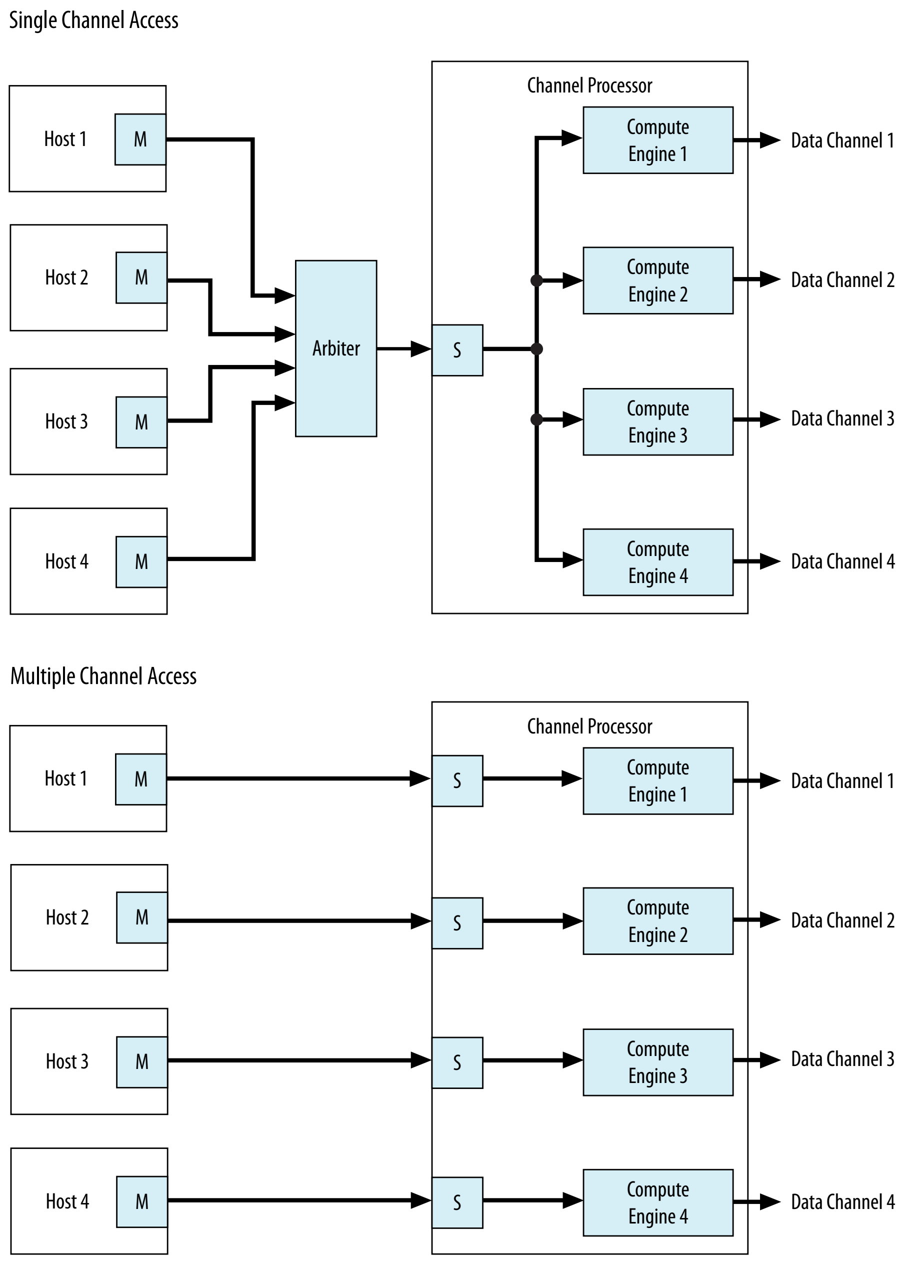
In this example, the DMA engine operates with a single master, because in AXI, the write and read channels on the master are independent and can process transactions simultaneously. There is concurrency between the read and write channels, with the yellow lines representing concurrent data paths.



Implementing Concurrency With Multiple Slaves

You can create multiple slave interfaces for a particular function to increase concurrency in your design.

Figure 7-6: Single Interface Versus Multiple Interfaces

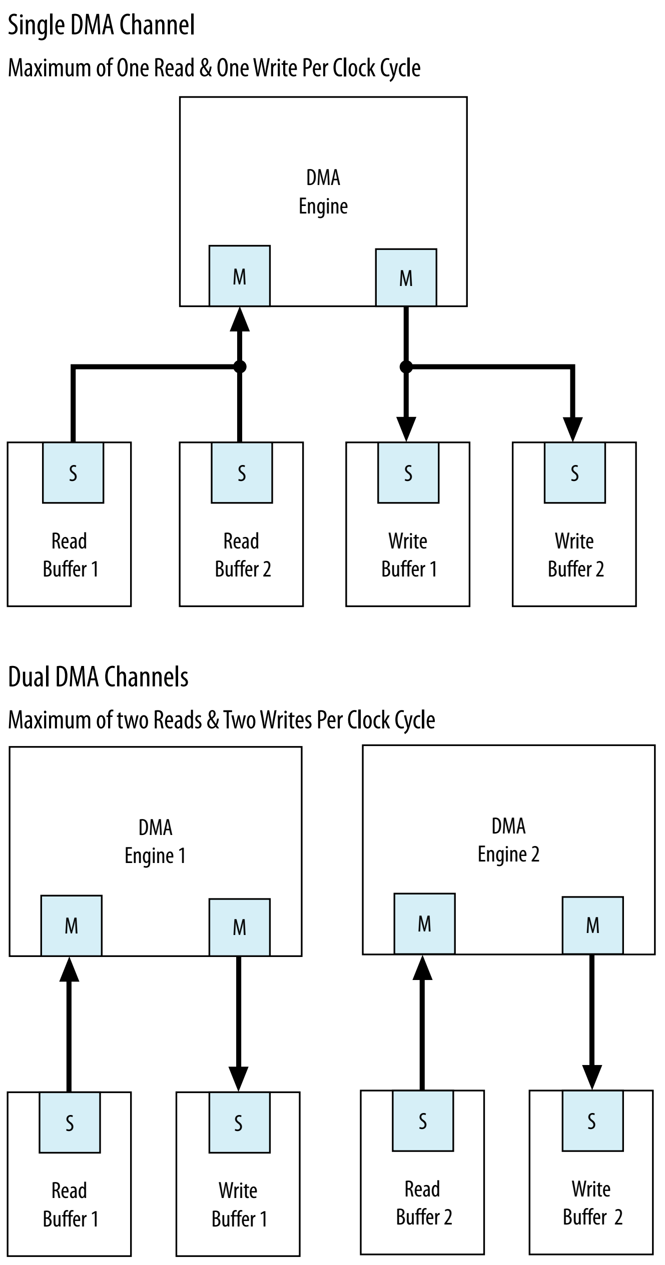


In this example, there are two channel processing systems. In the first, four hosts must arbitrate for the single slave interface of the channel processor. In the second, each host drives a dedicated slave interface, allowing all master interfaces to simultaneously access the slave interfaces of the 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 7-7: 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 4-1

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 4-1
- **Qsys Interconnect** on page 6-1

Using Bridges to Increase System Frequency

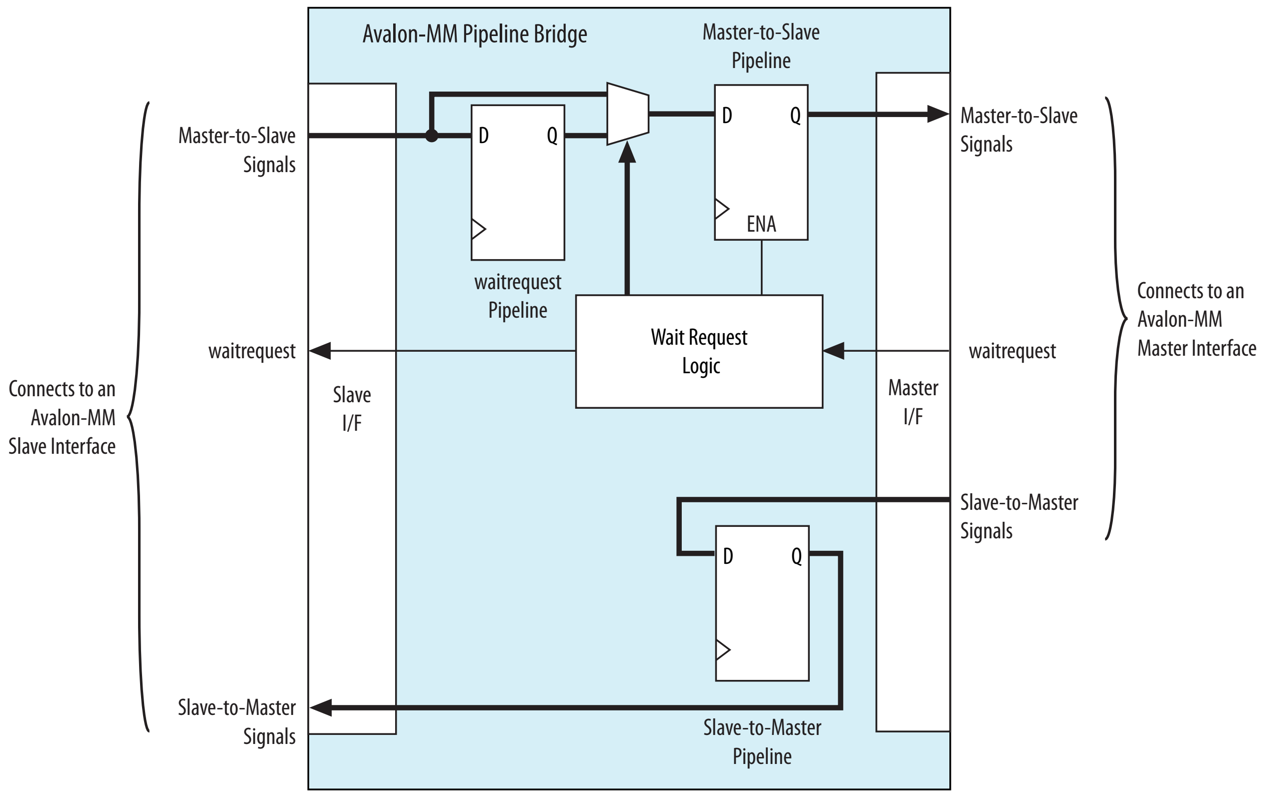
In Qsys, you can introduce interconnect pipeline stages or pipeline bridges to increase clock frequency in your system. Bridges control the system interconnect topology and allow you to subdivide the interconnect, giving you more control over pipelining and clock crossing functionality.

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 7-8: Avalon-MM Pipeline Bridge

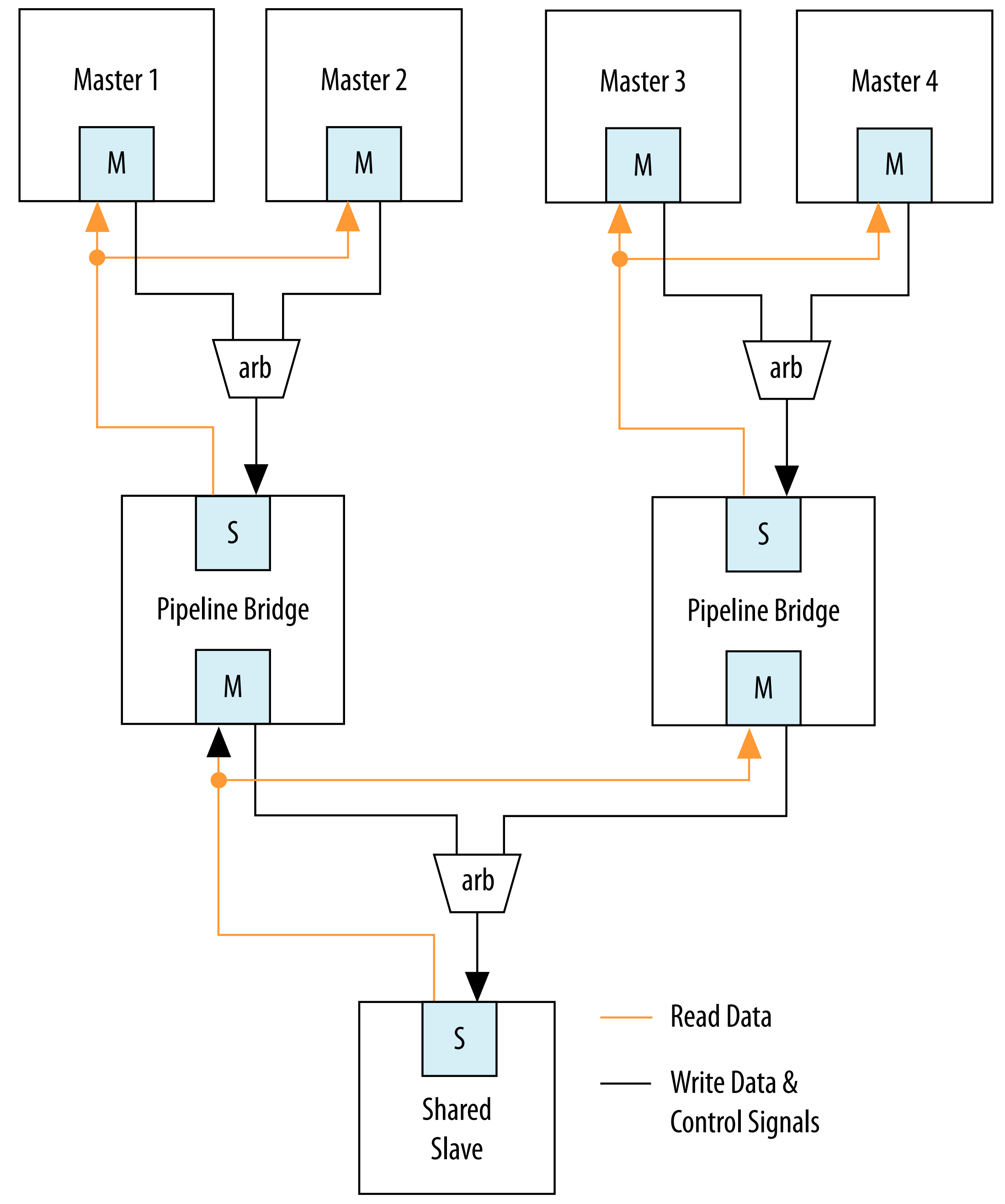


Implementing Command Pipelining (Master-to-Slave)

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 7-9: 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 Prime 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 Prime 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

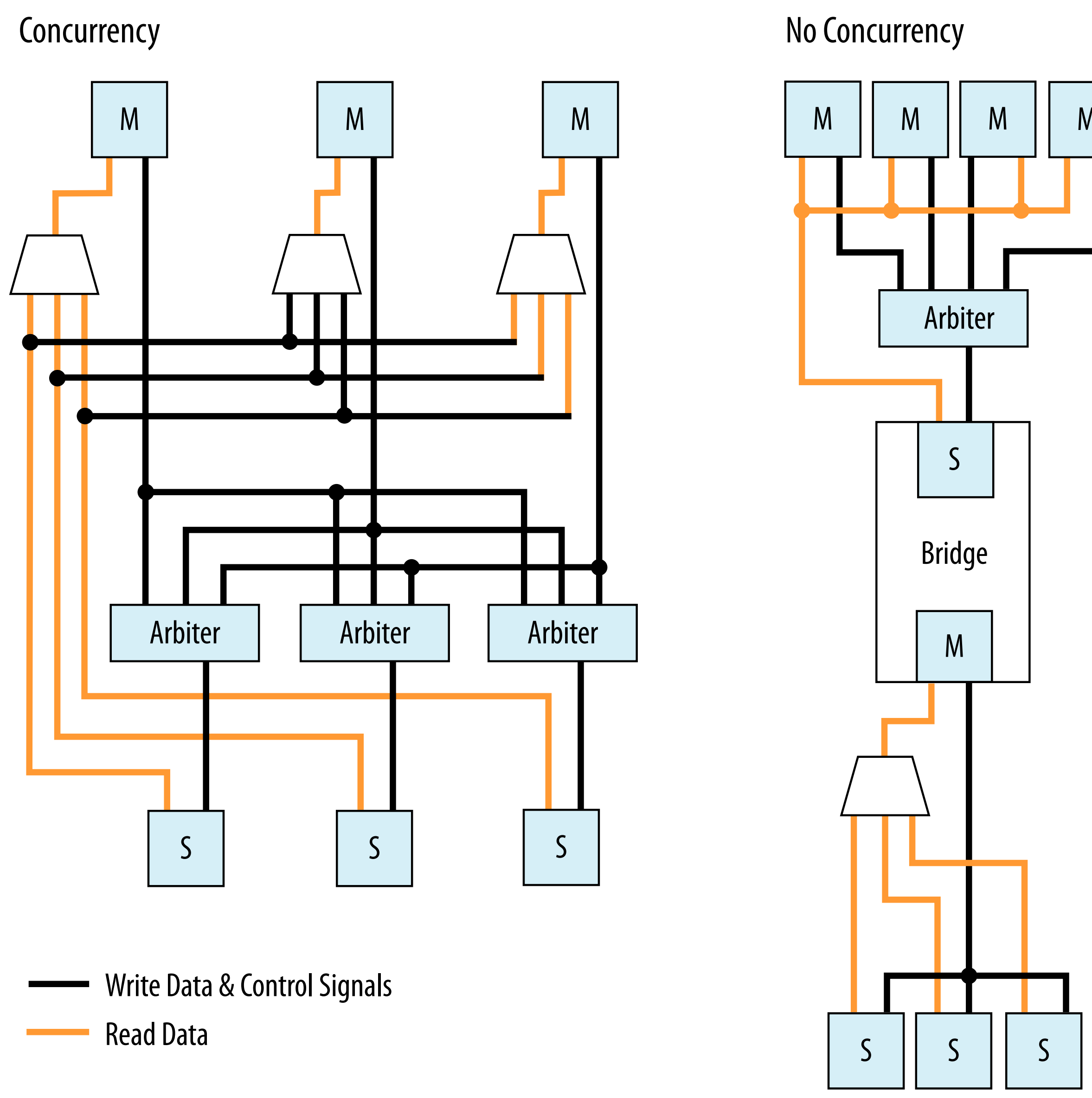
The amount of logic generated for the interconnect often increases as the system becomes larger because Qsys creates arbitration logic for every slave interface that is shared by multiple master interfaces. Qsys inserts multiplexer logic between master interfaces that connect to multiple slave interfaces 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 7-10: 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

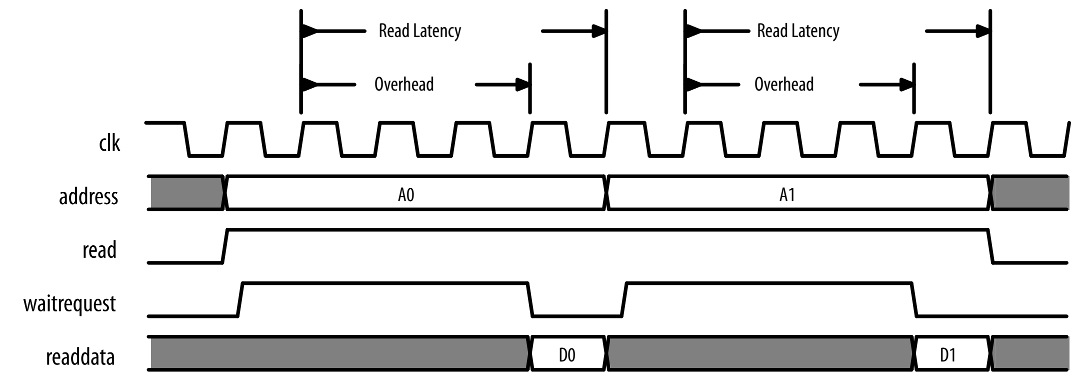
Adding a bridge to a design has an effect on the read latency between the master and the slave. Depending on the system requirements and the type of master and slave, this latency increase may or may not be acceptable in your design.

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.

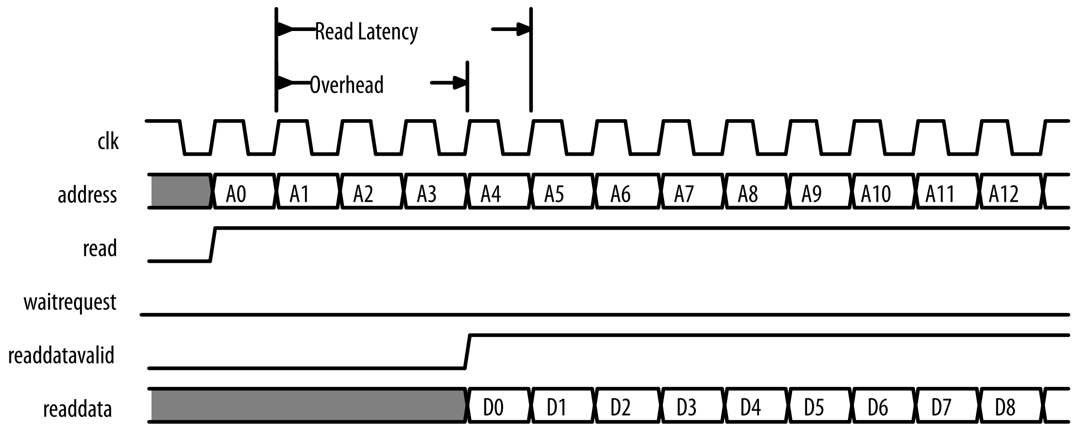
For example, if you use a pipelined read master such as a DMA controller to read data from a component with a fixed read latency of four clock cycles, but only perform a single word transfer, the overhead is three clock cycles out of the total of four. This is true when there is no additional pipeline latency in the interconnect. The read throughput is only 25%.

Figure 7-11: 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 7-12: High Efficiency Read Transfer

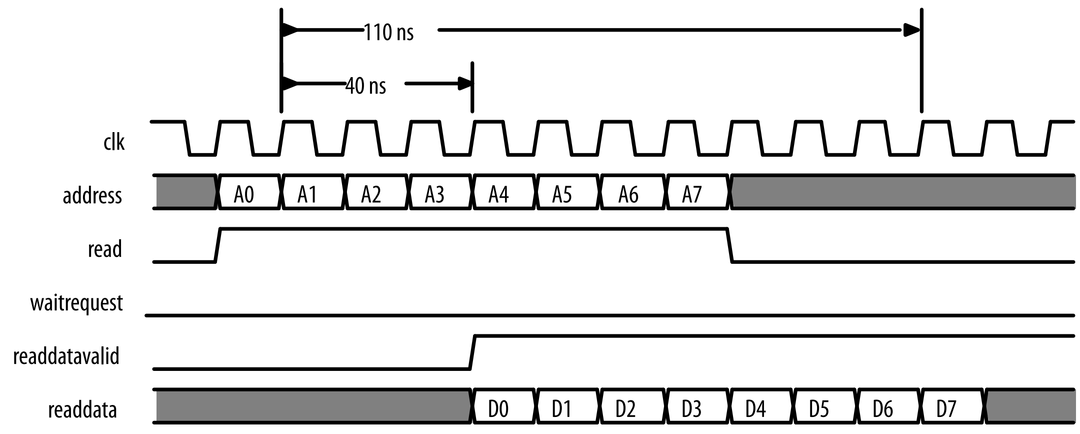


Unacceptable Latency Increase

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.

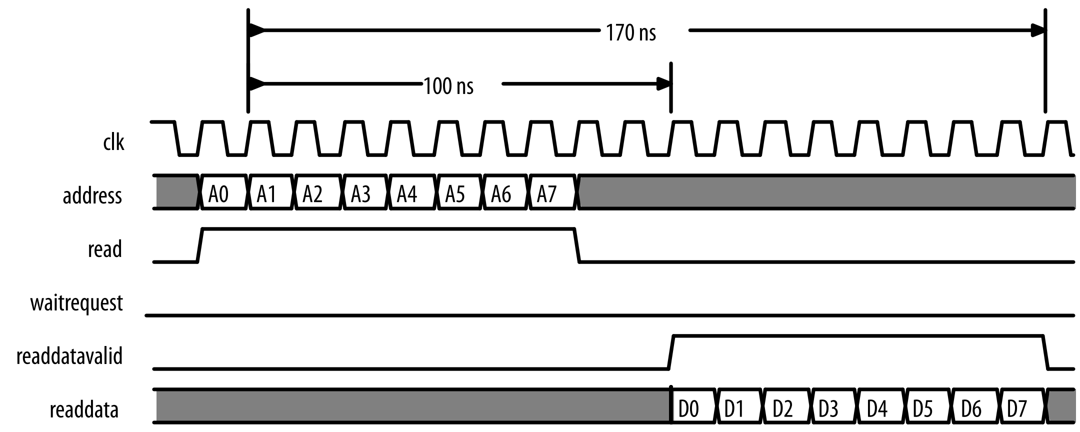
A Nios II processor instruction master has a cache memory with a read latency of four cycles, which is eight sequential words of data return for each read. At 100 MHz, the first read takes 40 ns to complete. Each successive word takes 10 ns so that eight reads complete in 110 ns.

Figure 7-13: 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 7-14: Performance of a Nios II Processor and Eight Reads with Ten Cycles Latency

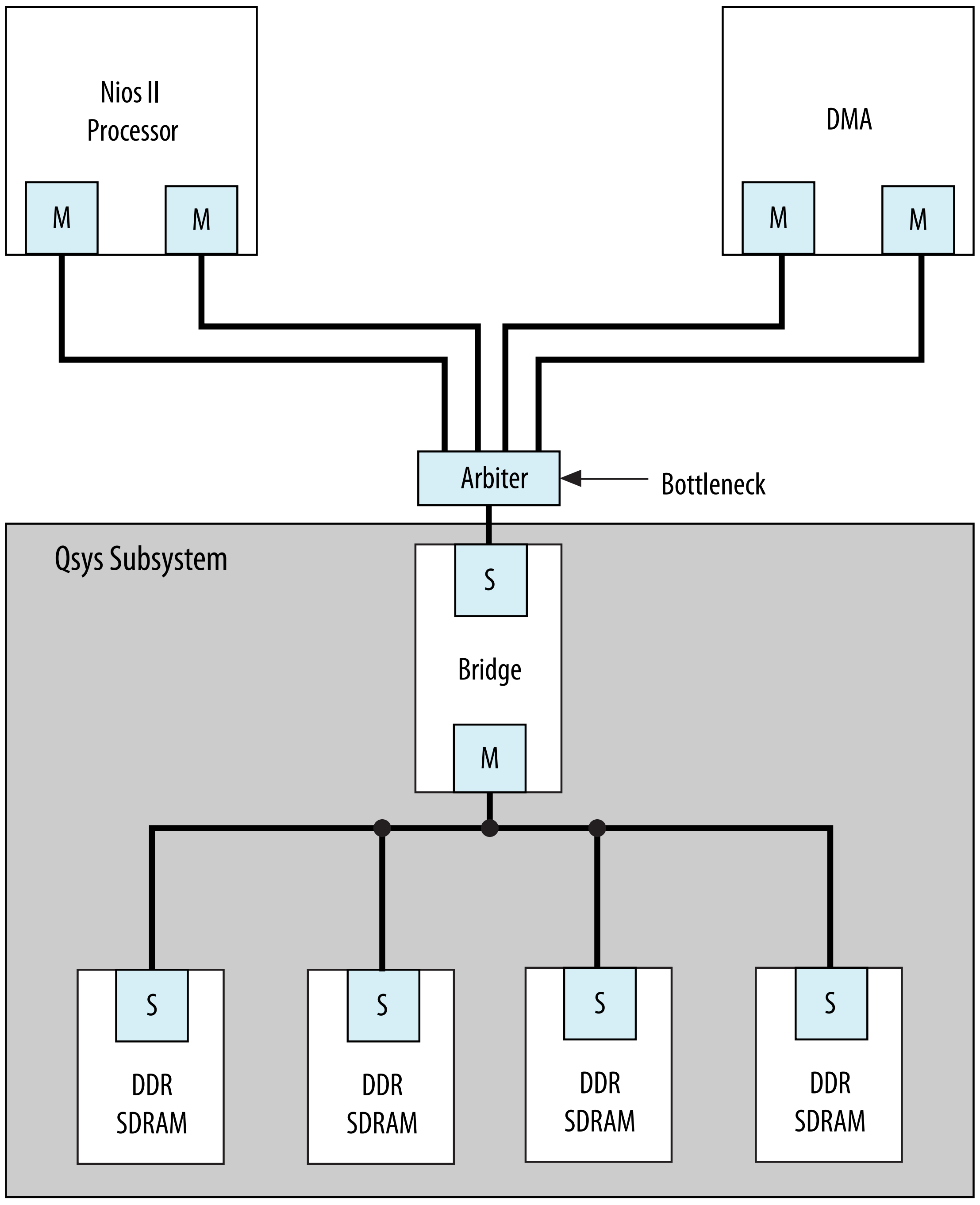


Limited Concurrency

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.

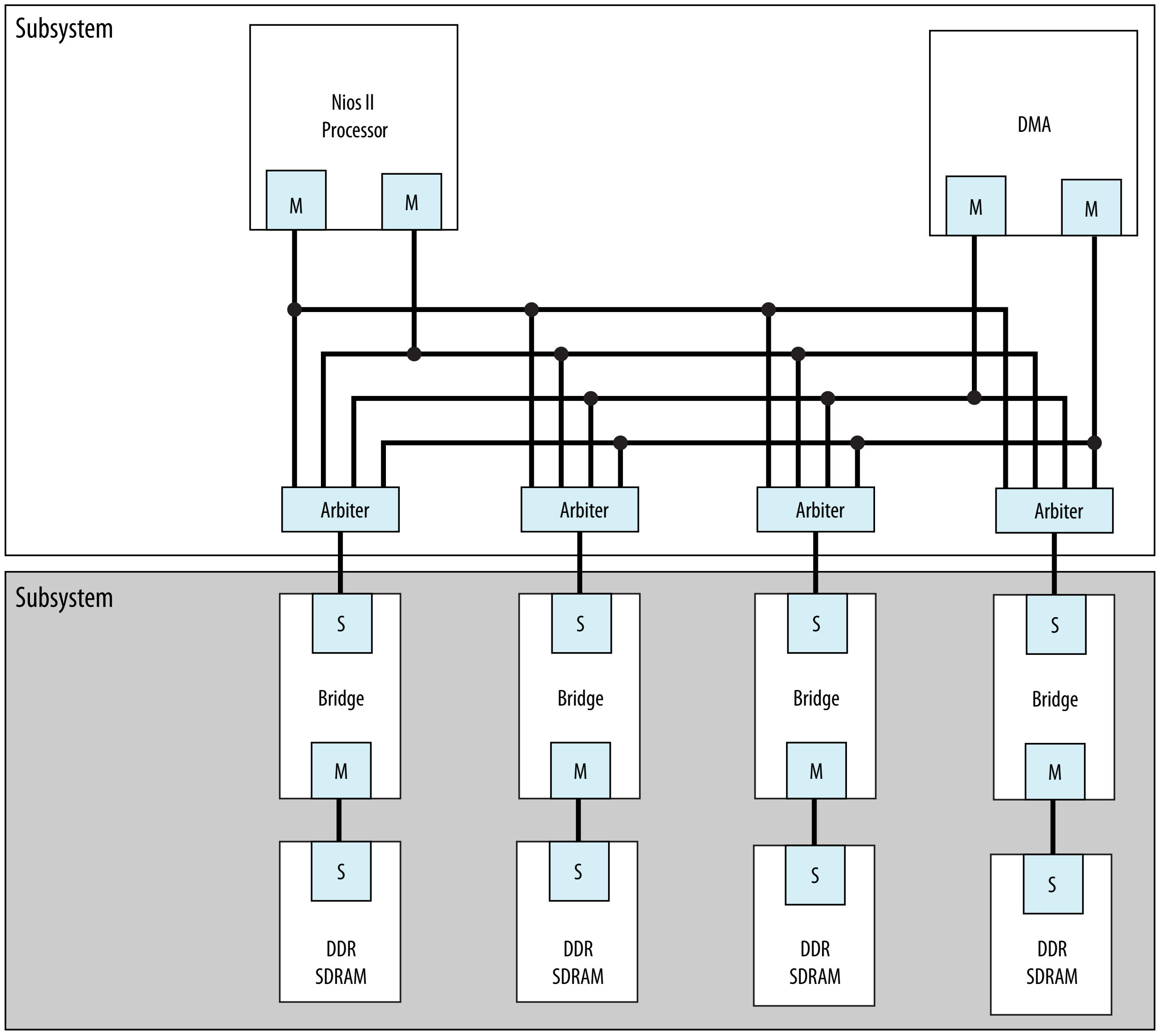
Figure 7-15: Inappropriate Use of a Bridge in a Hierarchical System



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 7-16: Efficient Memory Pipelining Without a Bottleneck in a Hierarchical System

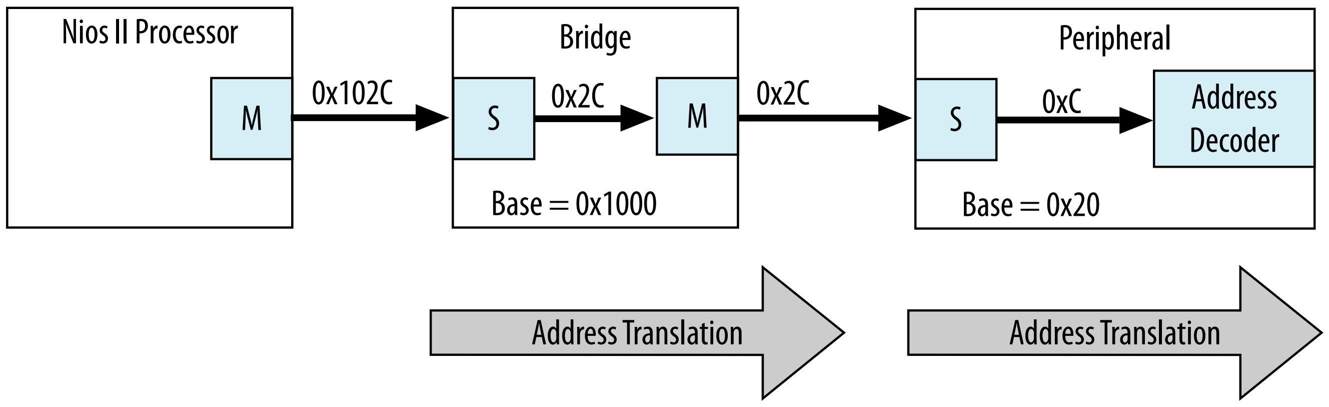


Address Space Translation

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 7-17: Bridge Address Translation

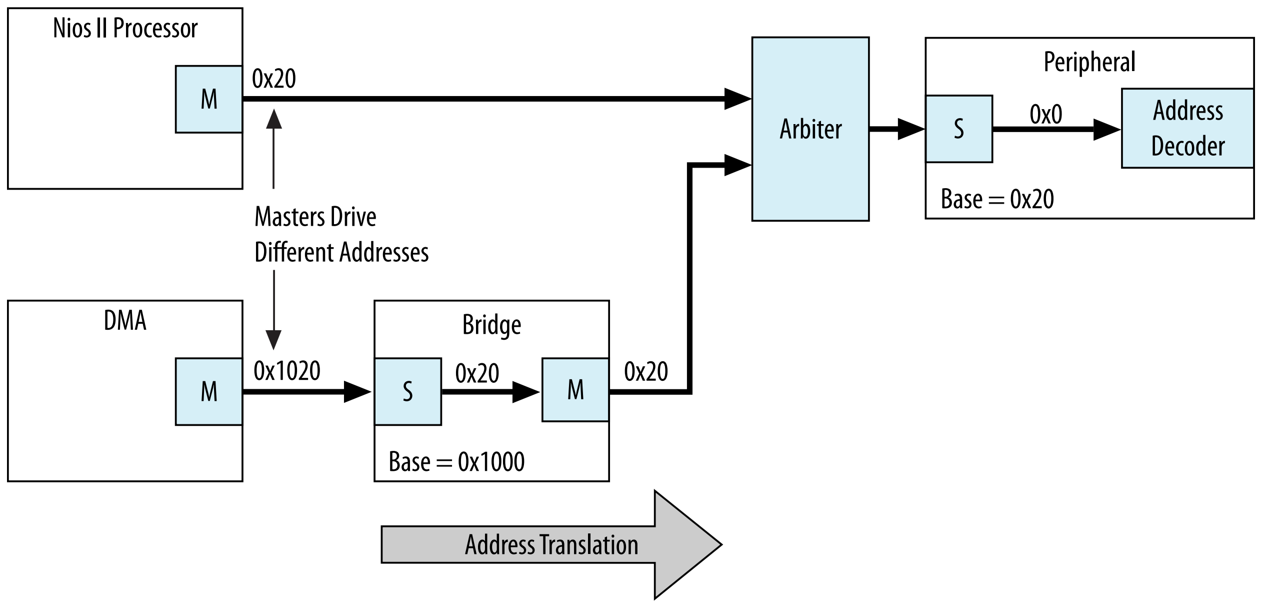


In this example, the Nios II processor connects to a bridge located at base address 0x1000, a slave connects to the bridge master interface at an offset of 0x20, and the processor performs a write transfer to the fourth 32-bit or 64-bit word within the slave. Nios II drives the address 0x102C to interconnect, 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.

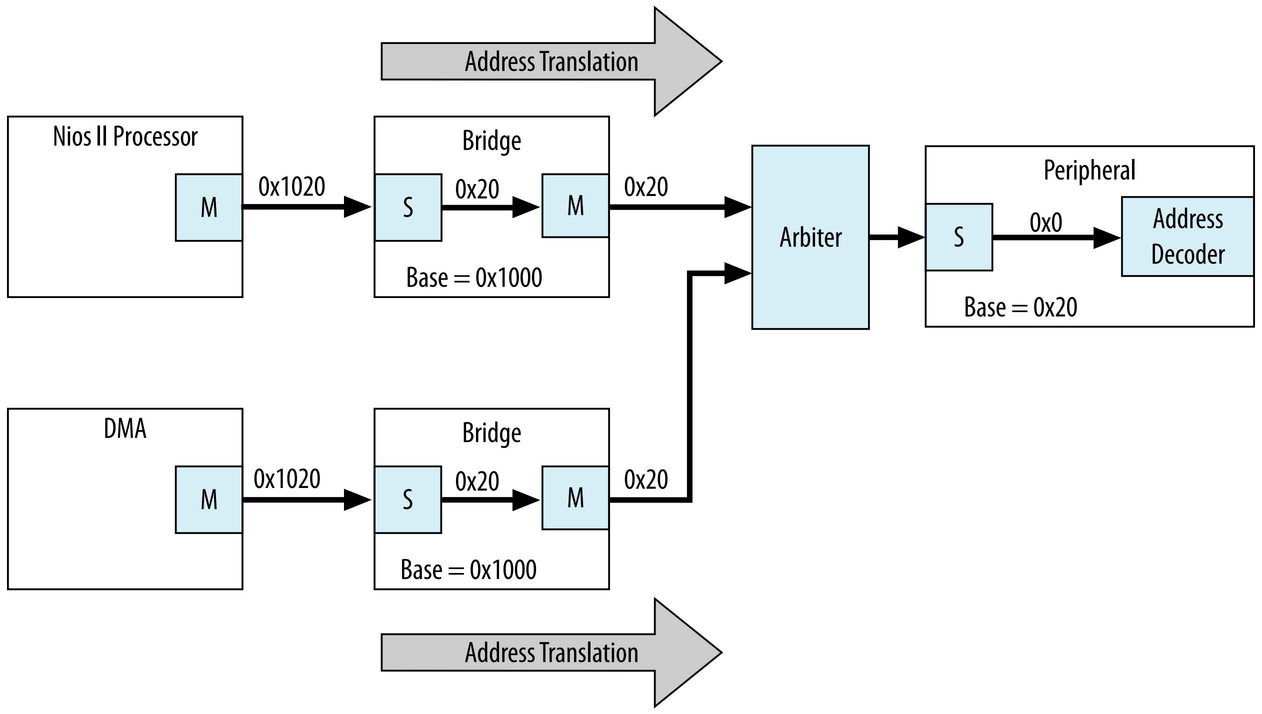
Figure 7-18: Slaves at Different Addresses and Complicating the System



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 7-19: 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 6-1

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 7-1: 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.

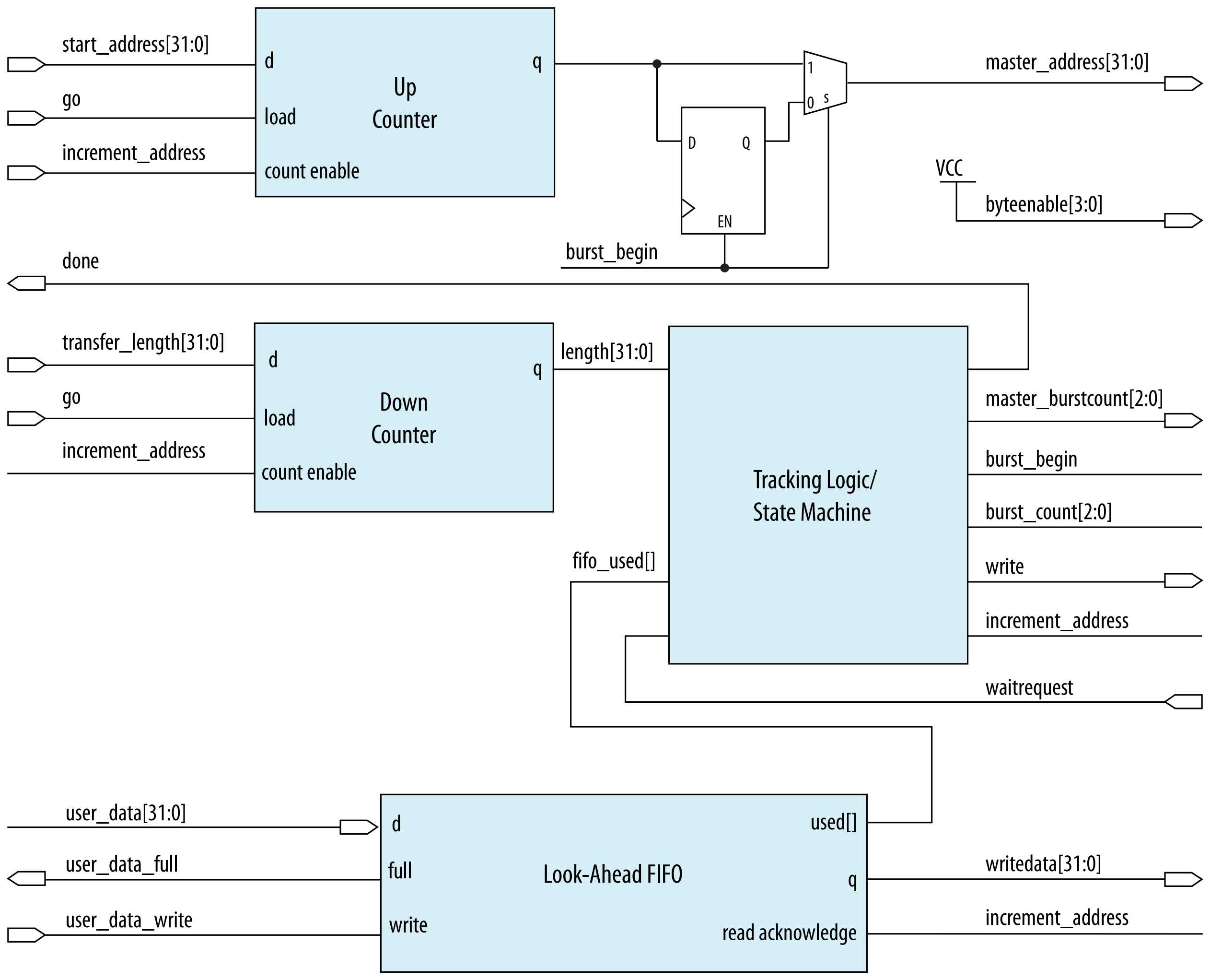
You can use a burst-capable slave interface if you know that your component requires sequential transfers to operate efficiently. Because SDRAM memories incur a penalty when switching banks or rows, performance improves when SDRAM memories are accessed sequentially with bursts.

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 7-20: 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 7-20

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 7-13

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

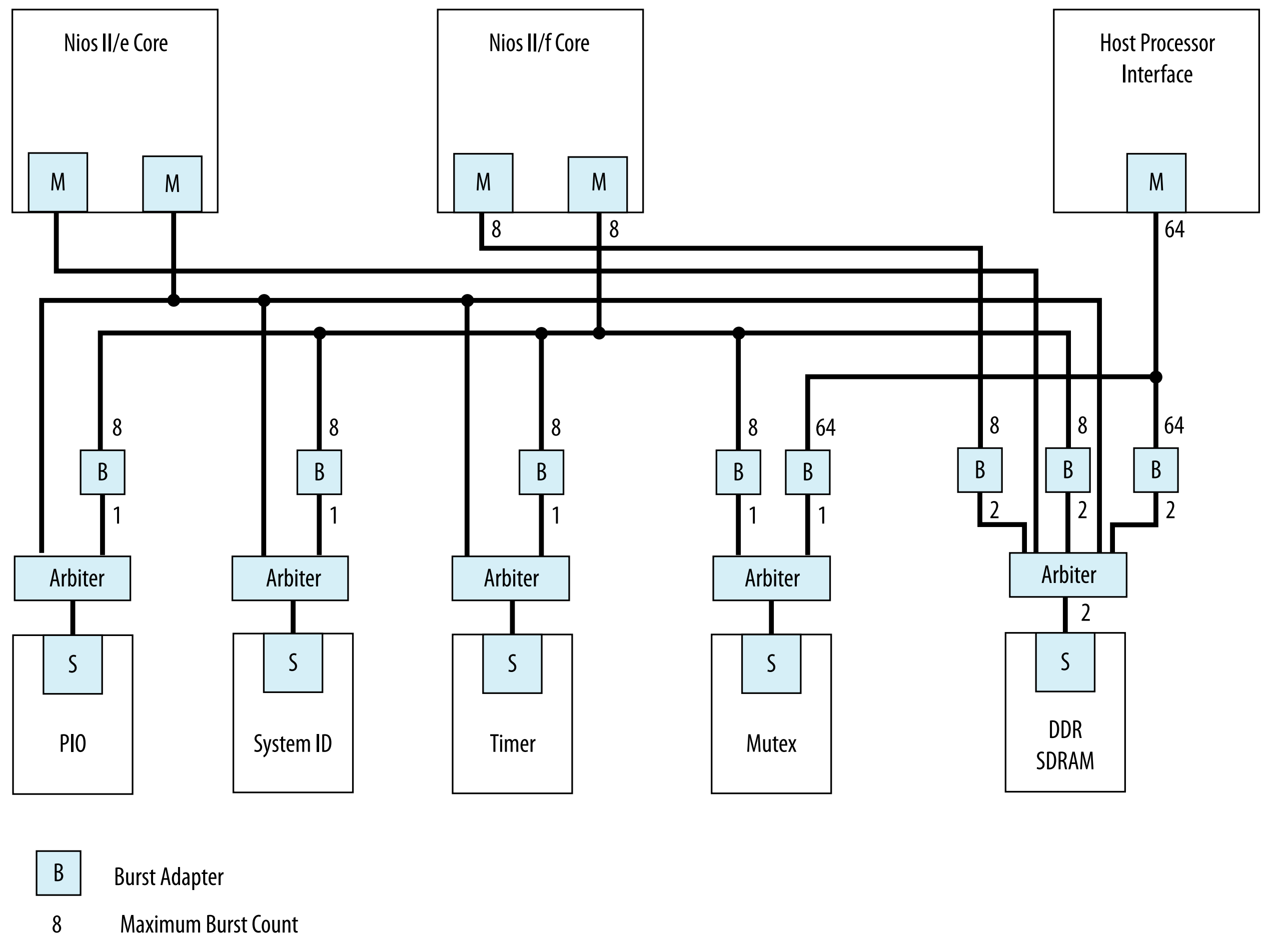
Using Concurrency in Memory-Mapped Systems on page 7-7

Consolidating Interfaces

In this example, we have a system with a mix of components, each having different burst capabilities: 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.

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 7-21: Mixed Bursting System

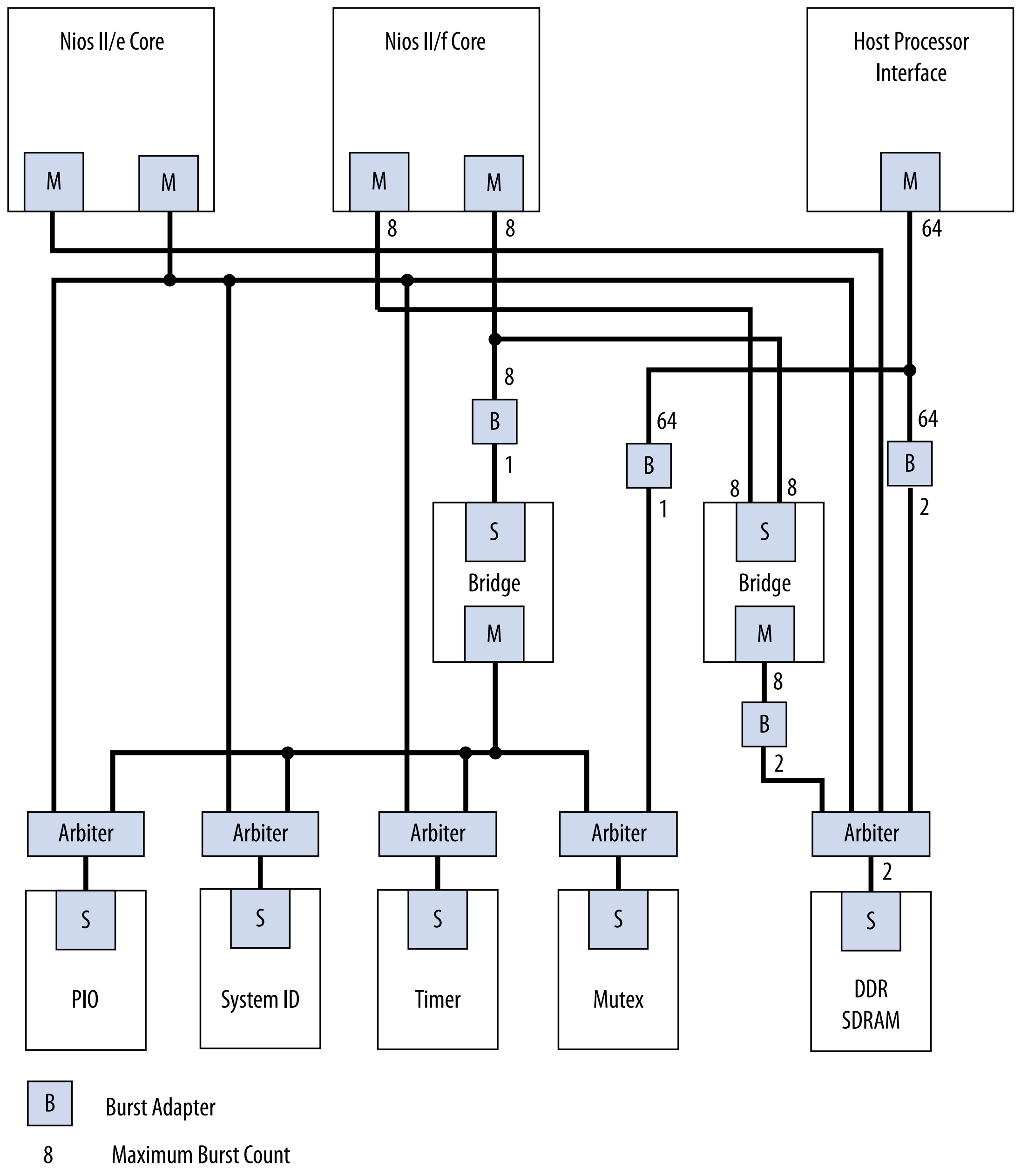


Qsys automatically inserts burst adapters to compensate for burst length mismatches. The adapters reduce bursts to a single transfer, or the length of two transfers. For the external processor interface connecting to DDR SDRAM, a burst of 64 words is divided into 32 burst transfers, each with a burst 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 7-22: 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.

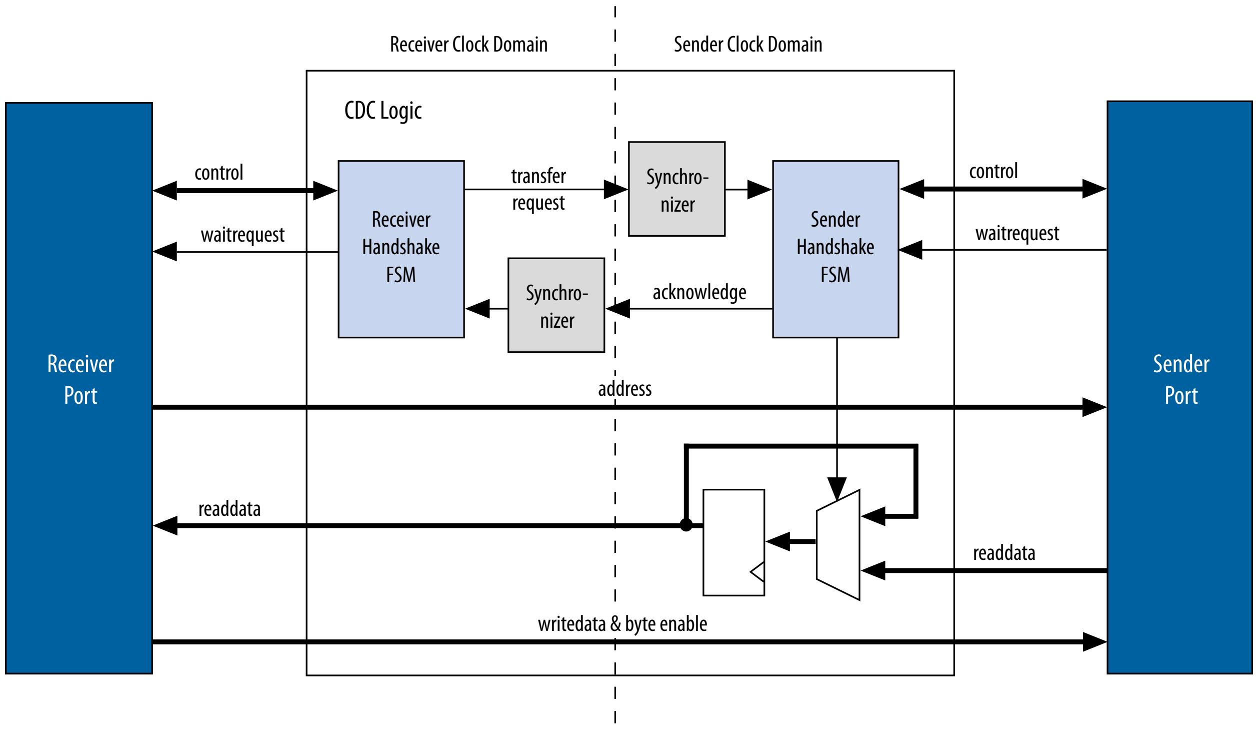
Qsys generates Clock Domain Crossing Logic (CDC) that hides the details of interfacing components operating in different clock domains. The interconnect supports the memory-mapped protocol with each port independently, and therefore masters do not need to incorporate clock adapters in order to interface 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 7-23: 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).

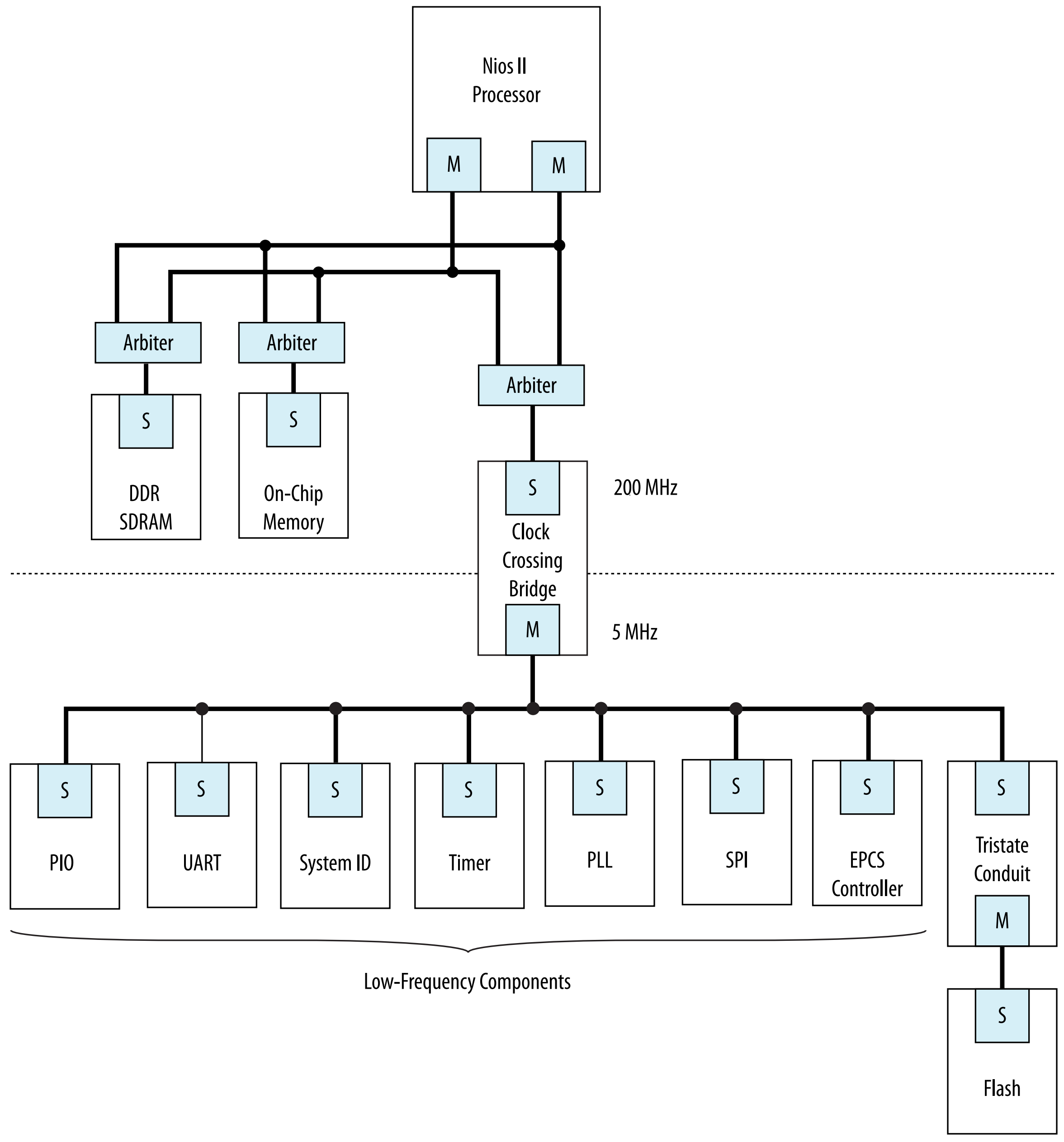
You can use clock crossing in Qsys to reduce the clock frequency of the logic that does not require a high frequency clock, which allows you to reduce power consumption. You can use either handshaking clock crossing bridges or handshaking clock crossing adapters to separate clock domains.

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 7-24: 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

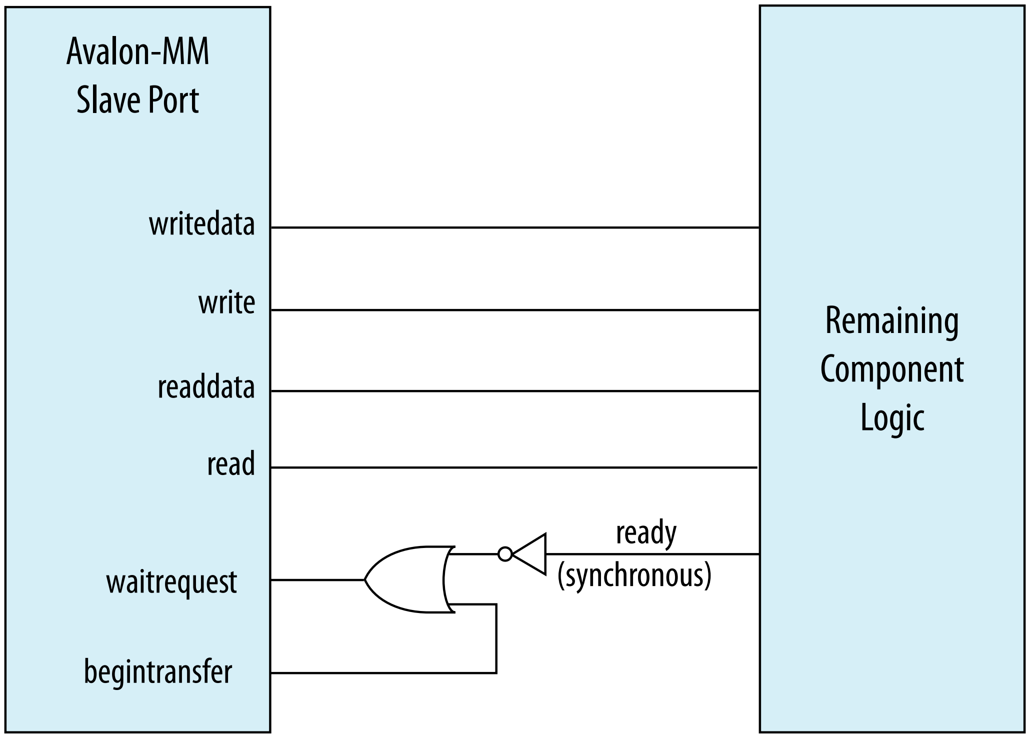
Qsys interconnect is uniquely combinational when no adapters or bridges are present and there is no interconnect pipelining. When a slave interface is not selected by a master, various signals may toggle and propagate into the component. By registering the boundary of your component at the master or 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 7-25: 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

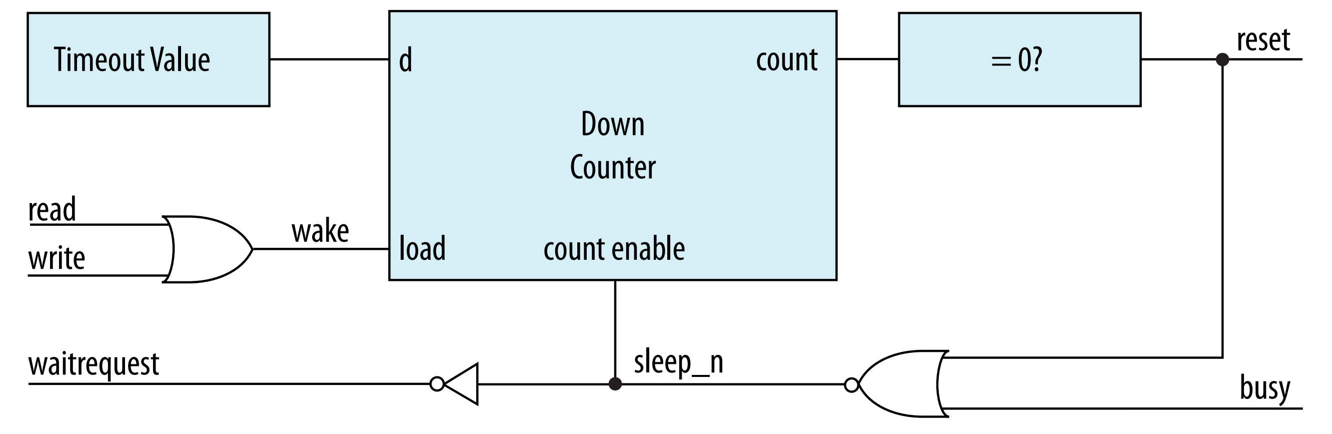
To design a component that supports software-controlled sleep mode, create a single memory-mapped location that enables and disables logic by writing a zero or one. You can use the register's output as a clock enable or reset, depending on whether the component has non-volatile requirements. The slave 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 7-26: 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

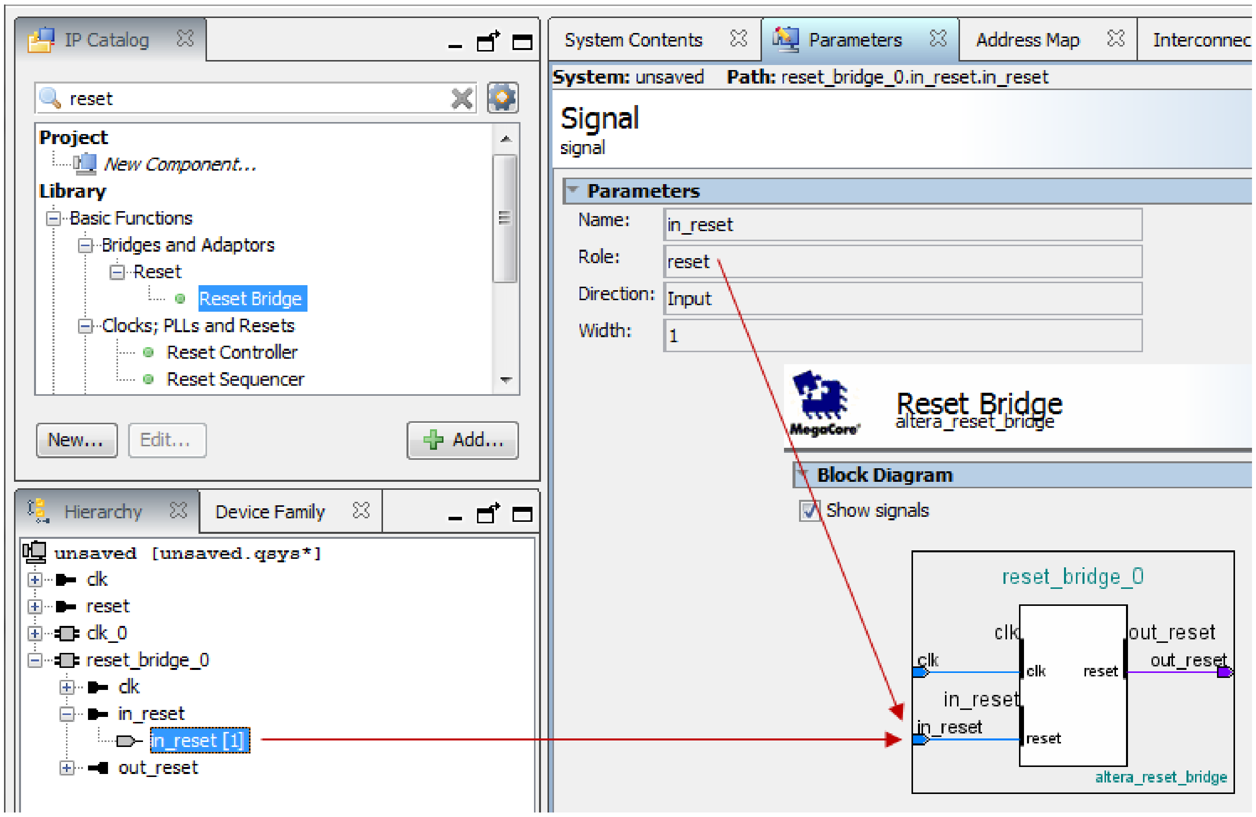
- Mutex Core
- Power Optimization

Reset Polarity and Synchronization in Qsys

When you add a component interface with a reset signal, Qsys defines its polarity as `reset`(active-high) or `reset_n` (active-low).

You can view the polarity status of a reset signal by selecting the signal in the **Hierarchy** tab, and then view its expanded definition in the open **Parameters** and **Block Symbol** tabs. When you generate your component, Qsys interconnect automatically inverts ploarities as needed.

Figure 7-27: Reset Signal (Active-High)



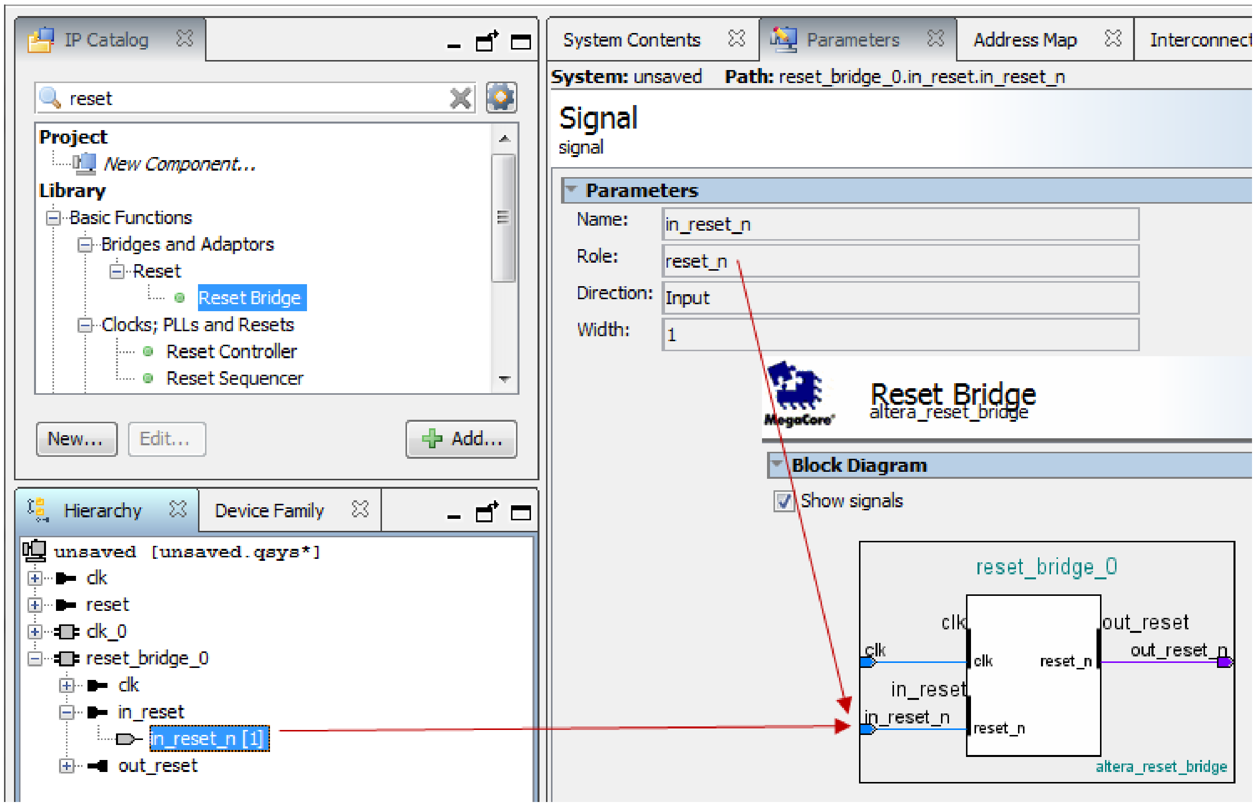
System: unsaved **Path:** reset_bridge_0.in_reset.in_reset

Signal
signal

Parameters

Name:	in_reset
Role:	reset
Direction:	Input
Width:	1

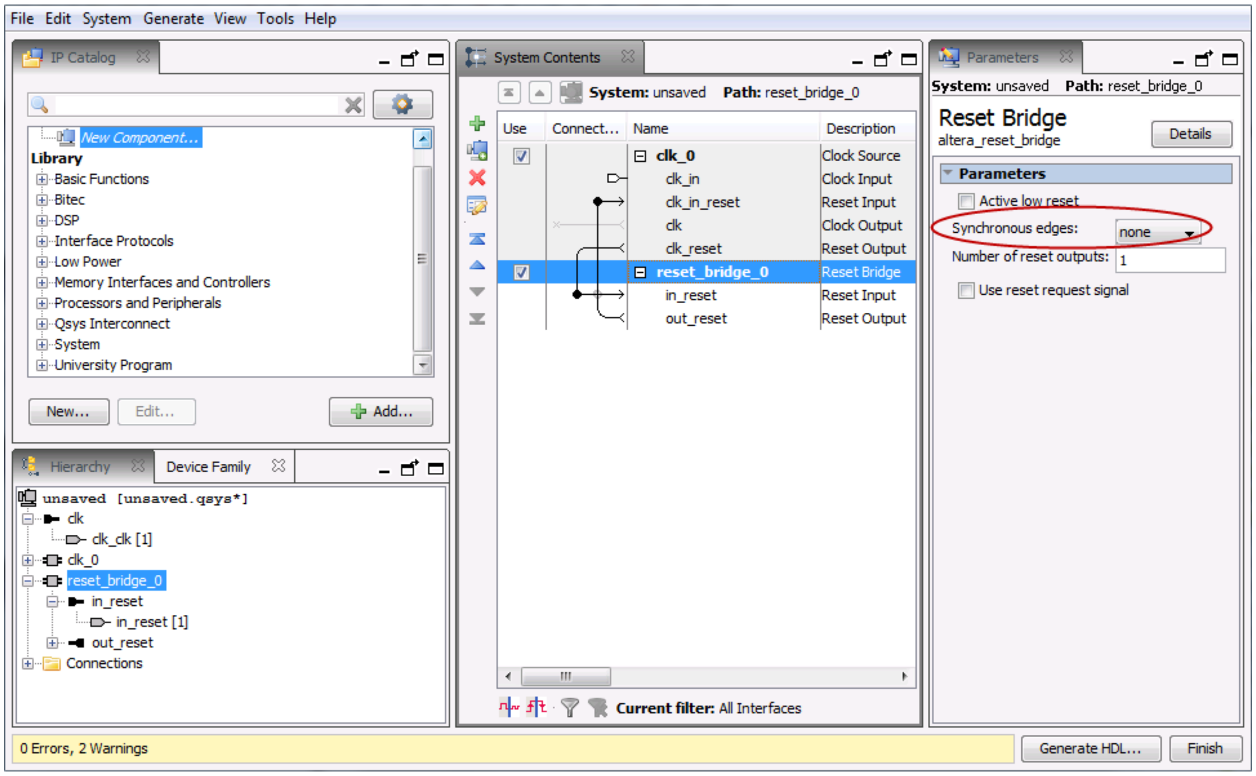
Figure 7-28: Reset Signal Active-Low



Each Qsys component has its own requirements for reset synchronization. Some blocks have internal synchronization and have no requirements, whereas other blocks require an externally synchronized reset. You can define how resets are synchronized in your Qsys system with the **Synchronous edges** parameter. In the clock source or reset bridge component, set the value of the **Synchronous edges** parameter to one of the following, depending on how the reset is externally synchronized:

- **None**—There is no synchronization on this reset.
- **Both**—The reset is synchronously asserted and deasserted with respect to the input clock.
- **Deassert**—The reset is synchronously asserted with respect to the input clock, and asynchronously deasserted.

Figure 7-29: Synchronous Edges Parameter



System: unsaved **Path:** reset_bridge_0

Use	Connect...	Name	Description
✓		clk_0	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
✓		reset_bridge_0	Reset Bridge
		in_reset	Reset Input
		out_reset	Reset Output

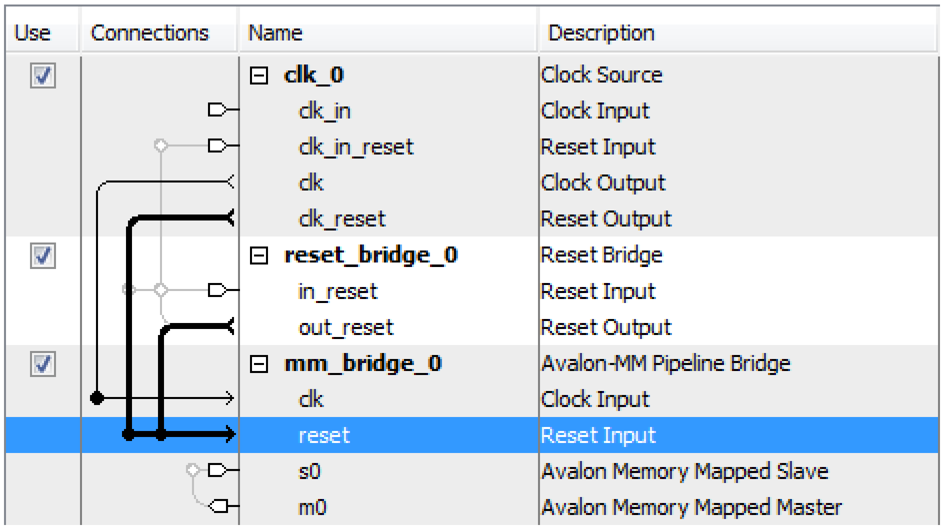
System: unsaved **Path:** reset_bridge_0

Reset Bridge
altera_reset_bridge

Parameters
- Active low reset
- Synchronous edges: none
- Number of reset outputs: 1
- Use reset request signal

You can combine multiple reset sources to reset a particular component.

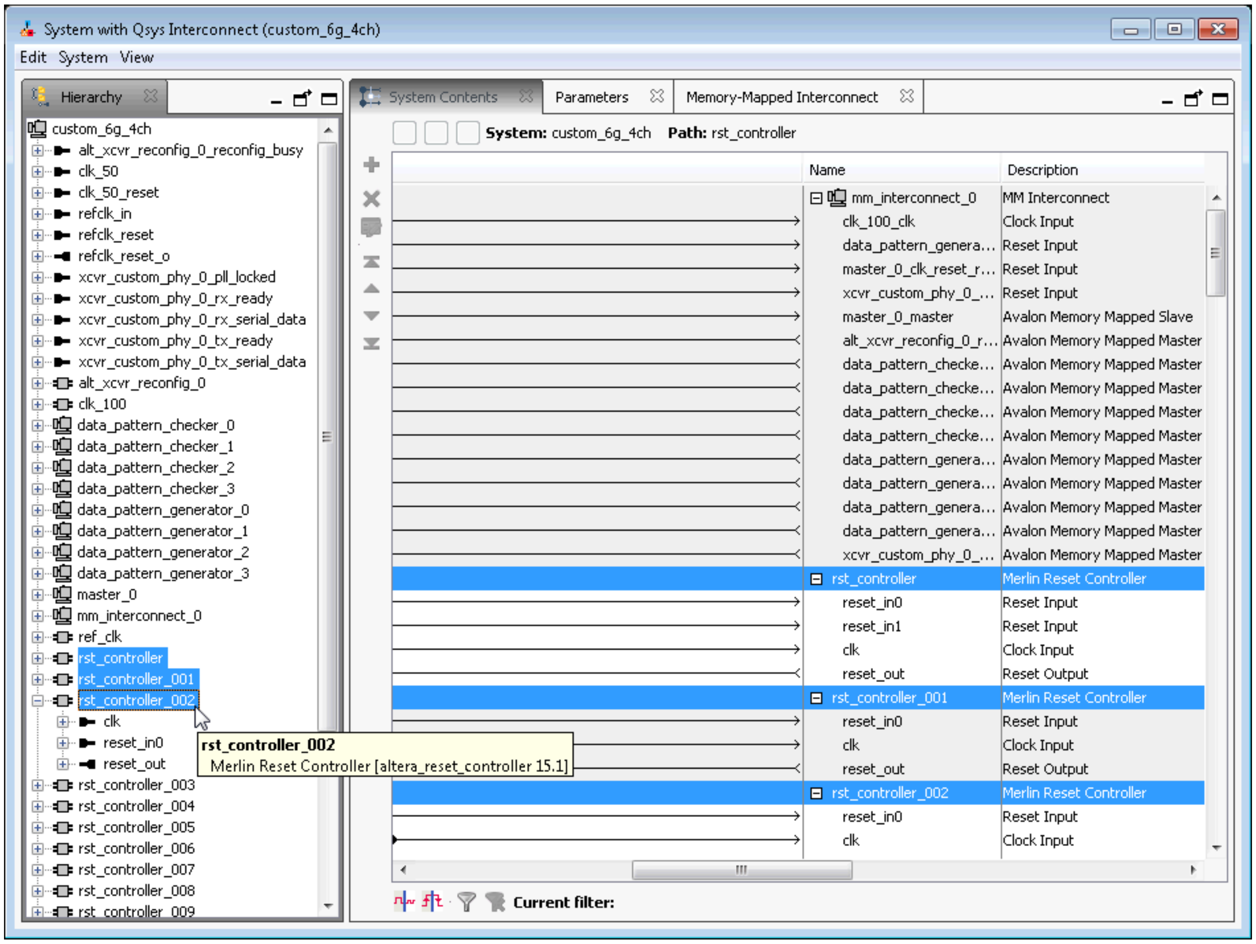
Figure 7-30: Combine Multiple Reset Sources



Use	Connections	Name	Description
✓		clk_0	Clock Source
		clk_in	Clock Input
		clk_in_reset	Reset Input
		clk	Clock Output
		clk_reset	Reset Output
✓		reset_bridge_0	Reset Bridge
		in_reset	Reset Input
		out_reset	Reset Output
✓		mm_bridge_0	Avalon-MM Pipeline Bridge
		clk	Clock Input
		reset	Reset Input
		s0	Avalon Memory Mapped Slave
		m0	Avalon Memory Mapped Master

When you generate your component, Qsys inserts adapters to synchronize or invert resets if there are mismatches in polarity or synchronization between the source and destination. You can view inserted adapters on the **Memory-Mapped Interconnect** tab with the **System** > **System Show System with Qsys Interconnect** command.

Figure 7-31: Qsys Interconnect


System with Qsys Interconnect (custom_6g_4ch)
Edit System View
Hierarchy
System Contents
Parameters
Memory-Mapped Interconnect
Current filter:

System: custom_6g_4ch **Path:** rst_controller

- custom_6g_4ch
  - alt_xcvr_reconfig_0_reconfig_busy
  - clk_50
  - clk_50_reset
  - refclk_in
  - refclk_reset
  - refclk_reset_o
  - xcvr_custom_phy_0_pll_locked
  - xcvr_custom_phy_0_rx_ready
  - xcvr_custom_phy_0_rx_serial_data
  - xcvr_custom_phy_0_tx_ready
  - xcvr_custom_phy_0_tx_serial_data
  - alt_xcvr_reconfig_0
  - clk_100
  - data_pattern_checker_0
  - data_pattern_checker_1
  - data_pattern_checker_2
  - data_pattern_checker_3
  - data_pattern_generator_0
  - data_pattern_generator_1
  - data_pattern_generator_2
  - data_pattern_generator_3
  - master_0
  - mm_interconnect_0
  - ref_clk
  - rst_controller
  - rst_controller_001
  - rst_controller_002
    - clk
    - reset_in0
    - reset_out
  - rst_controller_003
  - rst_controller_004
  - rst_controller_005
  - rst_controller_006
  - rst_controller_007
  - rst_controller_008
  - rst_controller_009

rst_controller_002
Merlin Reset Controller [altera_reset_controller 15.1]

Name	Description
mm_interconnect_0	MM Interconnect
clk_100_clk	Clock Input
data_pattern_genera...	Reset Input
master_0_clk_reset_r...	Reset Input
xcvr_custom_phy_0_...	Reset Input
master_0_master	Avalon Memory Mapped Slave
alt_xcvr_reconfig_0_r...	Avalon Memory Mapped Master
data_pattern_checke...	Avalon Memory Mapped Master
data_pattern_checke...	Avalon Memory Mapped Master
data_pattern_checke...	Avalon Memory Mapped Master
data_pattern_checke...	Avalon Memory Mapped Master
data_pattern_genera...	Avalon Memory Mapped Master
data_pattern_genera...	Avalon Memory Mapped Master
data_pattern_genera...	Avalon Memory Mapped Master
data_pattern_genera...	Avalon Memory Mapped Master
xcvr_custom_phy_0_...	Avalon Memory Mapped Master
rst_controller	Merlin Reset Controller
reset_in0	Reset Input
reset_in1	Reset Input
clk	Clock Input
reset_out	Reset Output
rst_controller_001	Merlin Reset Controller
reset_in0	Reset Input
clk	Clock Input
reset_out	Reset Output
rst_controller_002	Merlin Reset Controller
reset_in0	Reset Input
clk	Clock Input

Optimizing Qsys System Performance Design Examples

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.

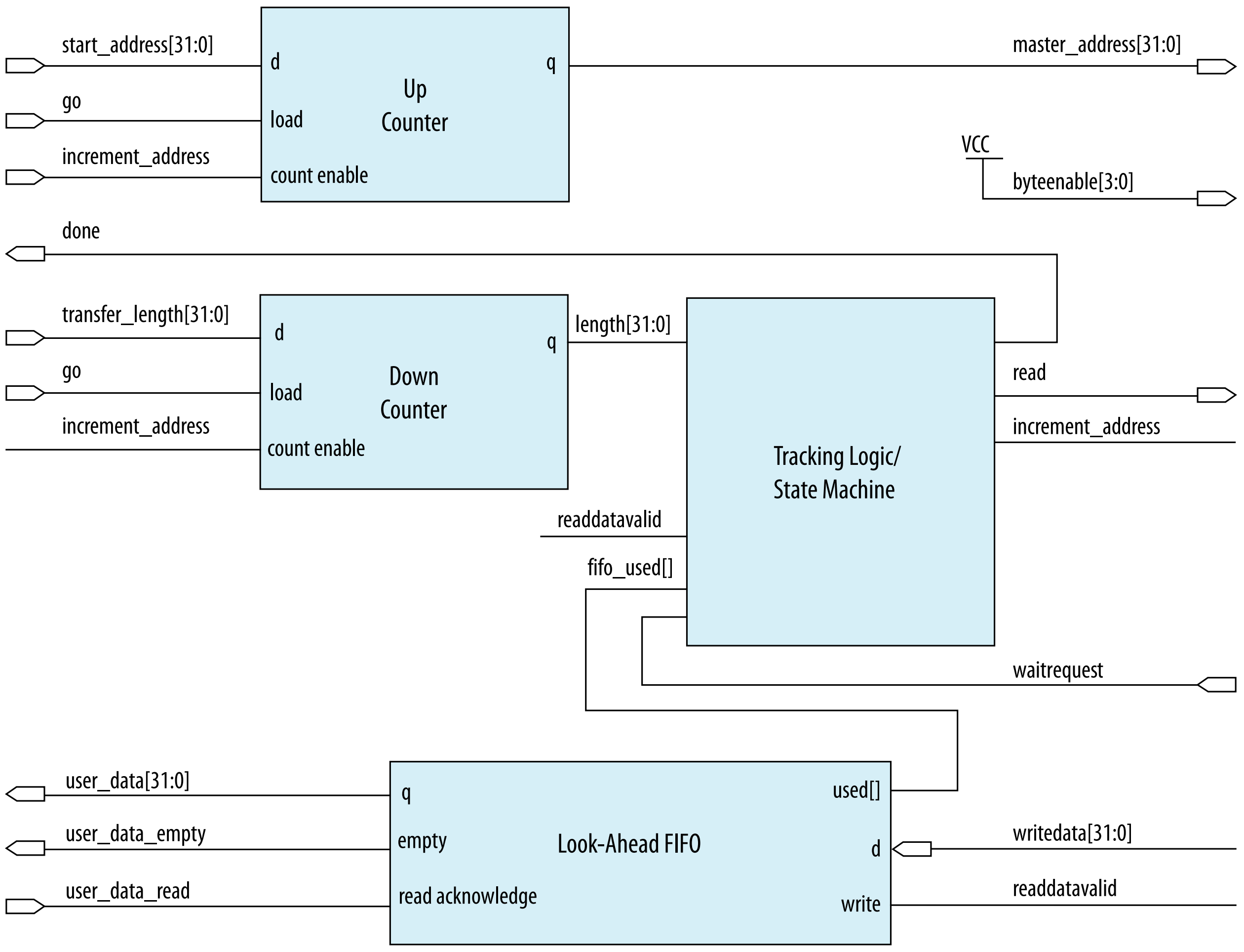
The data path logic includes the `readdata` and `readdatavalid` signals. If your master can accept data on every clock cycle, you can register the data with the `readdatavalid` as an enable bit. If your master cannot process a continuous stream of read data, it must buffer the data in a FIFO. The control 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 7-32: 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.

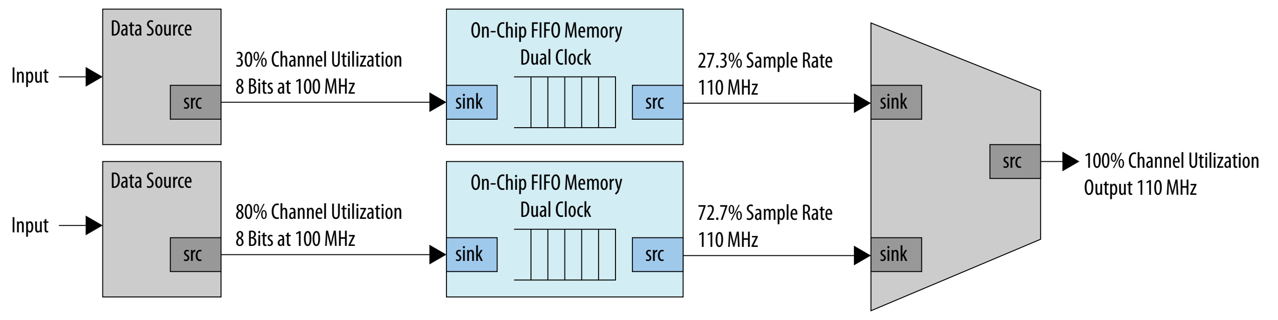
When the `length` register and the tracking logic counter reach zero, all the reads have completed and the done bit is asserted. The `done` bit is important if a second master overwrites the memory locations that the pipelined read master accesses. This bit guarantees that the reads have completed before 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.

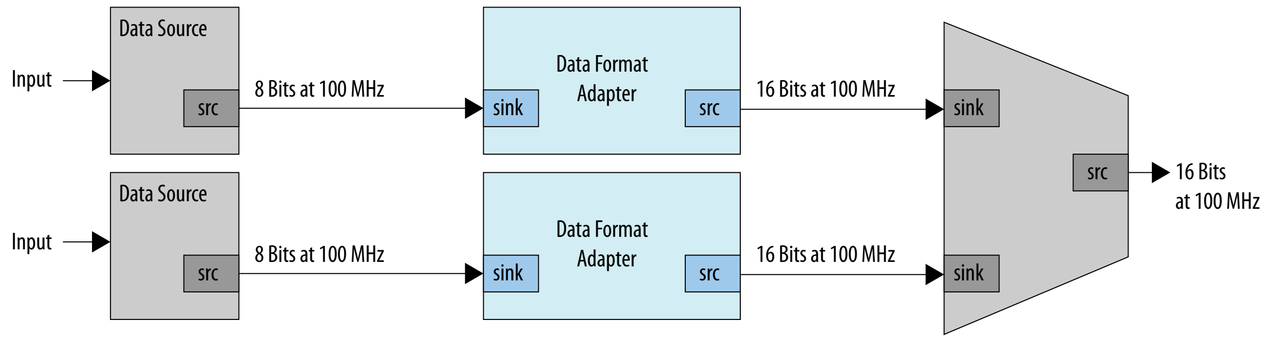
The diagram below illustrates a data path that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. The on-chip FIFO memory has an input clock frequency of 100 MHz, and an output clock 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 definitely 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 7-33: Data Path that Doubles the Clock Frequency



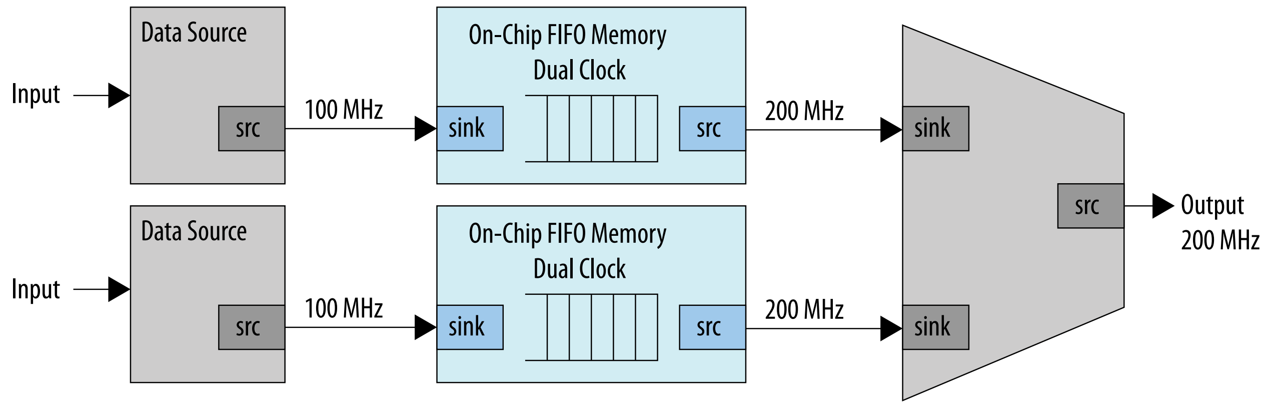
The diagram below illustrates a data path that uses a data format adapter and Avalon-ST channel multiplexer to merge the 8-bit 100 MHz input from two streaming data sources into a single 16-bit 100MHz streaming output. This example shows an output with double the throughput of each interface with a corresponding doubling of the data width.

Figure 7-34: 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 7-35: 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 7-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Added:*Reset Polarity and Synchronization in Qsys.* • Changed instances of *Quartus II* to *Quartus Prime.*

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

Component Interface Tcl Reference 8

2015.11.02

QPP5V1 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 Prime 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 5-1
- **Qsys Interconnect** on page 6-1
- **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 8-5
- **get_interface_assignments** on page 8-9
- **get_interface_properties** on page 8-11
- **get_interfaces** on page 8-7

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 8-3
- get_port_properties on page 8-13
- get_port_property on page 8-14

- get_port_property on page 8-14
- Direction Properties on page 8-99
- 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 8-3

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 8-3
- **get_interface_assignments** on page 8-9
- **get_interfaces** on page 8-7

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 8-3
- get_interface_assignment on page 8-8
- get_interfaces on page 8-7

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 8-5
- **get_port_property** on page 8-14
- **set_port_property** on page 8-18

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 8-12
- **set_interface_property** on page 8-17
- **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 8-11
- set_interface_property on page 8-17
- 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 8-5
- **get_port_property** on page 8-14
- **set_port_property** on page 8-18
- **Port Properties** on page 8-97

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 8-5
- **get_port_properties** on page 8-13
- **set_port_property** on page 8-18
- **Port Properties** on page 8-97

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.blockdia-gram.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 8-3
- **get_interface_assignment** on page 8-8
- **get_interface_assignments** on page 8-9

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 8-11
- **get_interface_property** on page 8-12
- **Interface Properties** on page 8-90
- **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 8-5
- **get_port_properties** on page 8-13
- **set_port_property** on page 8-18

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 8-23
- **get_parameter_property** on page 8-24
- **get_parameter_value** on page 8-25
- **set_parameter_property** on page 8-29
- **set_parameter_value** on page 8-30
- **Parameter Type Properties** on page 8-95

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 8-21
- **get_parameter_property** on page 8-24
- **get_parameter_value** on page 8-25
- **get_parameters** on page 8-22
- **set_parameter_property** on page 8-29

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 8-21
- **get_parameter_property** on page 8-24
- **get_parameter_value** on page 8-25
- **get_parameters** on page 8-22
- **set_parameter_property** on page 8-29
- **Parameter Properties** on page 8-92

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 8-21
- **get_parameter_properties** on page 8-23
- **get_parameter_value** on page 8-25
- **get_parameters** on page 8-22
- **set_parameter_property** on page 8-29
- **set_parameter_value** on page 8-30
- **Parameter Properties** on page 8-92

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 8-21
- **get_parameter_property** on page 8-24
- **get_parameters** on page 8-22
- **set_parameter_property** on page 8-29
- **set_parameter_value** on page 8-30

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 8-28

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 8-26
- **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 8-21
- **get_parameter_properties** on page 8-23
- **set_parameter_property** on page 8-29
- **Parameter Properties** on page 8-92

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 8-36
- **get_display_item_property** on page 8-37
- **get_display_items** on page 8-35
- **set_display_item_property** on page 8-38
- **Display Item Kind Properties** on page 8-101

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 8-33
- get_display_item_properties on page 8-36
- get_display_item_property on page 8-37
- set_display_item_property on page 8-38

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 8-33
- **get_display_item_property** on page 8-37
- **set_display_item_property** on page 8-38
- **Display Item Properties** on page 8-100

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 8-33
- **get_display_item_properties** on page 8-36
- **get_display_items** on page 8-35
- **set_display_item_property** on page 8-38
- **Display Item Properties** on page 8-100

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 8-33
- **get_display_item_properties** on page 8-36
- **get_display_item_property** on page 8-37
- **Display Item Properties** on page 8-100

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 8-42
- **set_module_assignment** on page 8-47

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 8-41
- **set_module_assignment** on page 8-47

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 8-3
- add_interface_port on page 8-5

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 8-45
- **set_module_property** on page 8-48
- **Module Properties** on page 8-103

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 8-44
- **set_module_property** on page 8-48
- **Module Properties** on page 8-103

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 8-41
- **get_module_assignments** on page 8-42

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 8-44
- **get_module_property** on page 8-45
- **Module Properties** on page 8-103

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 8-67
- **get_instance_parameters** on page 8-65
- **get_instances** on page 8-57
- **set_instance_parameter_value** on page 8-70

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 8-52
- **get_instance_interface_property** on page 8-64
- **get_instance_parameter_value** on page 8-67
- **get_instance_parameters** on page 8-65
- **get_instance_property** on page 8-61
- **get_instances** on page 8-57
- **set_instance_parameter_value** on page 8-70

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 8-52
- get_instance_interfaces on page 8-58

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 8-52

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 parameter names

Arguments

connection

The connection to query.

Example

```
get_connection_parameters cpu.data_master/dma0.csr
```

Related Information

- **add_connection** on page 8-52
- **get_connection_parameter_value** on page 8-56

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 8-52
- **get_connection_parameters** on page 8-55

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 8-49
- add_instance on page 8-52
- get_instance_parameter_value on page 8-67
- get_instance_parameters on page 8-65
- set_instance_parameter_value on page 8-70

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 8-52
- **get_instance_interface_ports** on page 8-59
- **get_instance_interfaces** on page 8-58

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 8-52
- **get_instance_interfaces** on page 8-58
- **get_instance_port_property** on page 8-68

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 8-52
- **get_instance_interface_property** on page 8-64
- **get_instance_interfaces** on page 8-58

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 8-52
- **get_instance_properties** on page 8-63
- **set_instance_property** on page 8-62
- **Instance Properties** on page 8-91

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 8-52
- **get_instance_properties** on page 8-63
- **get_instance_property** on page 8-61
- **Instance Properties** on page 8-91

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 8-52
- **get_instance_property** on page 8-61
- **set_instance_property** on page 8-62
- **Instance Properties** on page 8-91

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 8-52
- get_instance_interfaces on page 8-58

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 8-49
- **add_instance** on page 8-52
- **get_instance_parameter_value** on page 8-67
- **get_instances** on page 8-57
- **set_instance_parameter_value** on page 8-70

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 8-52
- **Parameter Properties** on page 8-92

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 8-49
- **add_instance** on page 8-52
- **get_instance_parameters** on page 8-65
- **get_instances** on page 8-57
- **set_instance_parameter_value** on page 8-70

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 8-52
- **get_instance_interface_ports** on page 8-59
- **Port Properties** on page 8-97

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 8-52
- get_connection_parameter_value on page 8-56

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 8-49
- add_instance on page 8-52
- get_instance_parameter_value on page 8-67
- get_instances on page 8-57

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 8-73
- **get_fileset_property** on page 8-78
- **Fileset Properties** on page 8-105

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 Prime 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 8-72
- **get_fileset_file_attribute** on page 8-75
- **File Kind Properties** on page 8-109
- **File Source Properties** on page 8-110
- **File Attribute Properties** on page 8-108

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 8-72
- **Fileset Properties** on page 8-105

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 8-72
- **add_fileset_file** on page 8-73
- **get_fileset_file_attribute** on page 8-75
- **File Attribute Properties** on page 8-108

- **add_fileset** on page 8-72
- **add_fileset_file** on page 8-73
- **get_fileset_file_attribute** on page 8-75
- **File Attribute Properties** on page 8-108

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 8-72
- **get_fileset_properties** on page 8-77
- **set_fileset_property** on page 8-74
- **Fileset Properties** on page 8-105

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 8-105

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 8-72
- **set_fileset_sim_properties** on page 8-80
- **Operating System Properties** on page 8-117
- **Simulator Properties** on page 8-111

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 8-79
- **Operating System Properties** on page 8-117
- **Simulator Properties** on page 8-111

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 8-72
- **add_fileset_file** on page 8-73

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 8-84

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 8-83

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 8-86

set_qip_strings

Description

Places strings in the Quartus Prime IP File (**.qip**) file, which Qsys passes to the command as a Tcl list. You add the **.qip** file to your Quartus Prime 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 8-85

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 8-113
- **Parameter Type Properties** on page 8-95
- **Units Properties** on page 8-116

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 8-99

- **Port VHDL Type Properties** on page 8-112
- **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 Prime 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 Prime 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 Prime 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 8-115

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
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
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 Handbook Archive

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

Qsys System Design Components 9

2015.11.02

QPP5V1 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 4-1
- **Qsys Interconnect** on page 6-1
- **Embedded Peripherals IP User Guide**

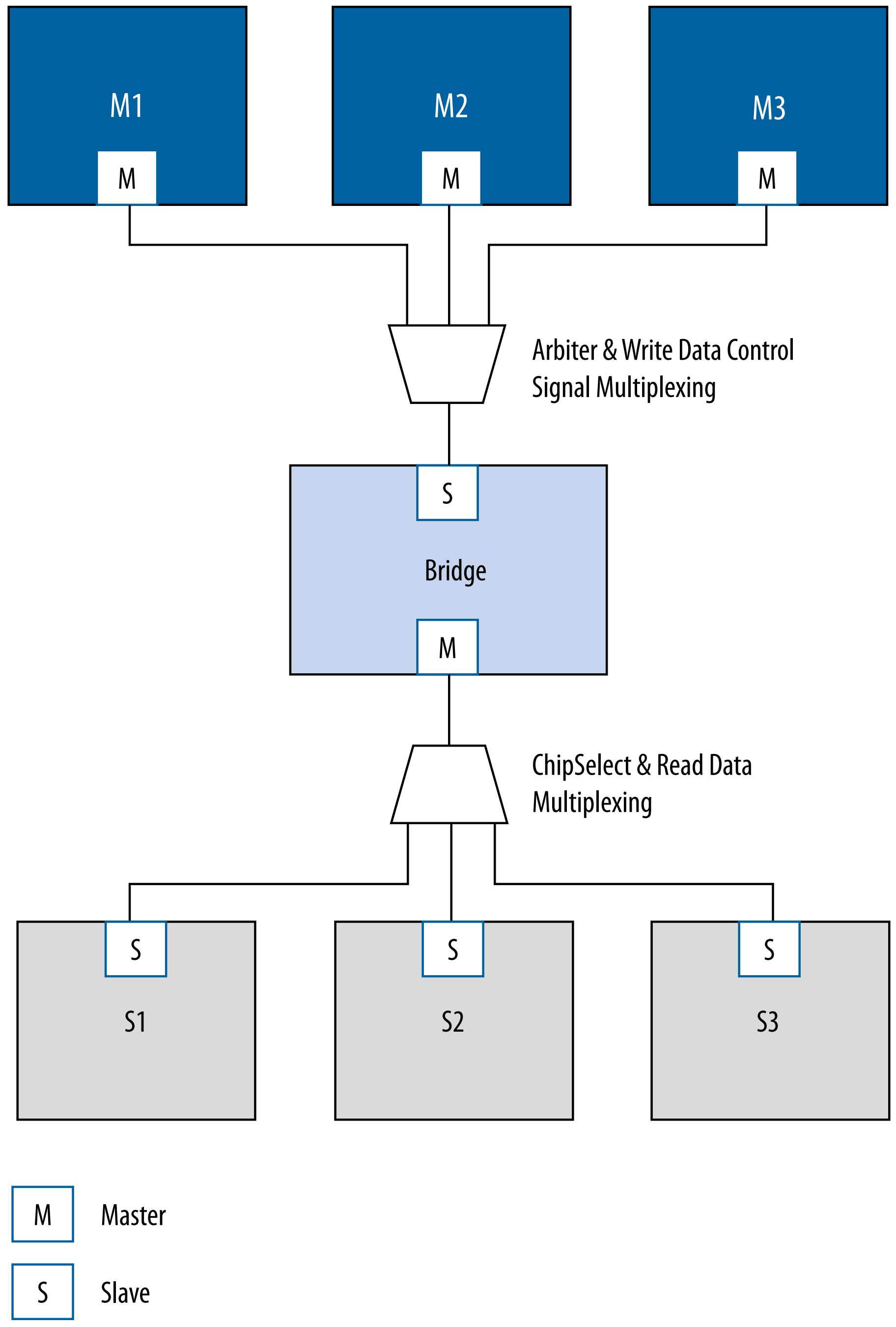
Bridges

Bridges affect the way Qsys transports data between components. You can insert bridges between master 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 9-1: 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 the transfers are initiated on the slave.

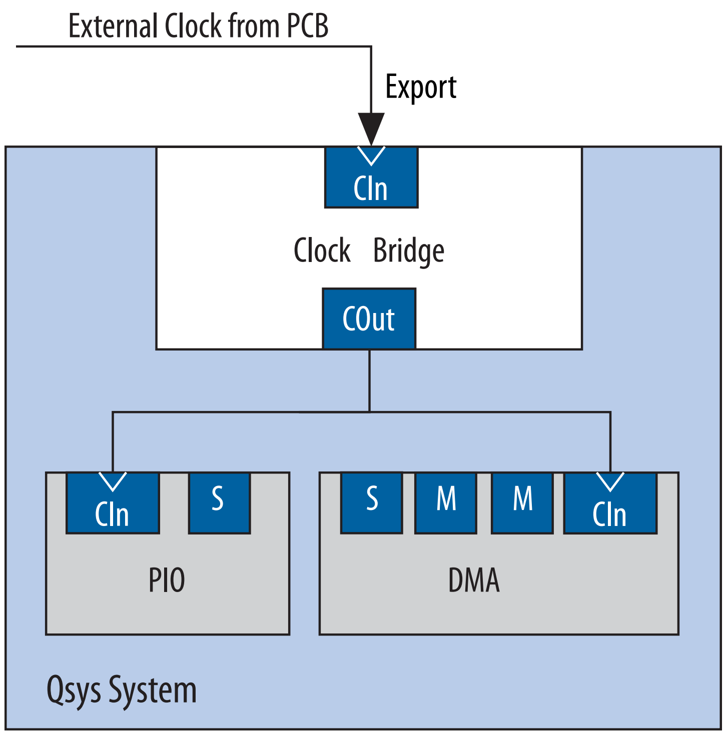

M1
M
M2
M
M3
M
Arbiter & Write Data Control Signal Multiplexing
S
Bridge
M
ChipSelect & Read Data Multiplexing
S
S1
S
S2
S
S3
M
Master
S
Slave

Clock Bridge

The Clock Bridge connects 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. Create the connection through an exported interface, and then connect to multiple clock input interfaces.

Clock outputs match fan-out performance 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 9-2: 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 connect to the reset sources for the connected master and slave components on either side of the DC FIFO. For this configuration, you must assert both the resets on the master and the slave sides at the same time to ensure the DC FIFO resets properly. Alternatively, you can drive both resets from the same reset source to guarantee that the DC FIFO resets properly.

Note: The clock crossing bridge includes appropriate SDC constraints for its internal asynchronous FIFOs. For these SDC constraints to work correctly, do 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; the split has the same effect as setting false paths.

Related Information

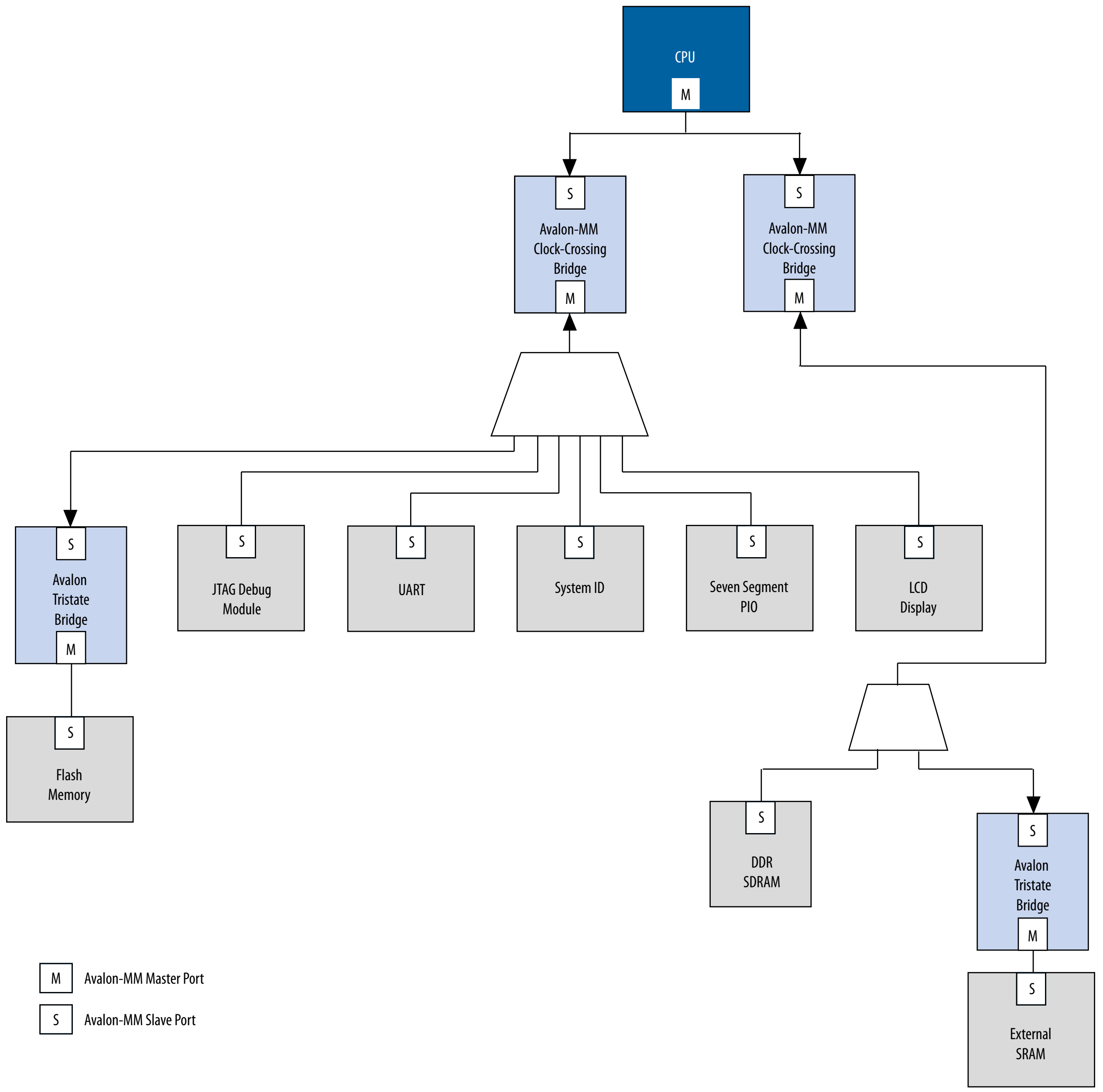
- **Creating a System with Qsys** on page 4-1

Avalon-MM Clock Crossing Bridge Example

In the example shown below, the Avalon-MM Clock Crossing bridges separate slave components into two groups. The Avalon-MM Clock Crossing Bridge places the low performance slave components behind a single bridge and clocks the components at a lower speed. The bridge places the high performance components behind a second bridge and clocks it at a higher speed.

By inserting clock-crossing bridges, you simplify the Qsys interconnect and allow the Quartus Prime Fitter to optimize paths that require minimal propagation delay.

Figure 9-3: Avalon-MM Clock Crossing Bridge



Avalon-MM Clock Crossing Bridge Parameters

Table 9-1: 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	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 mean time 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

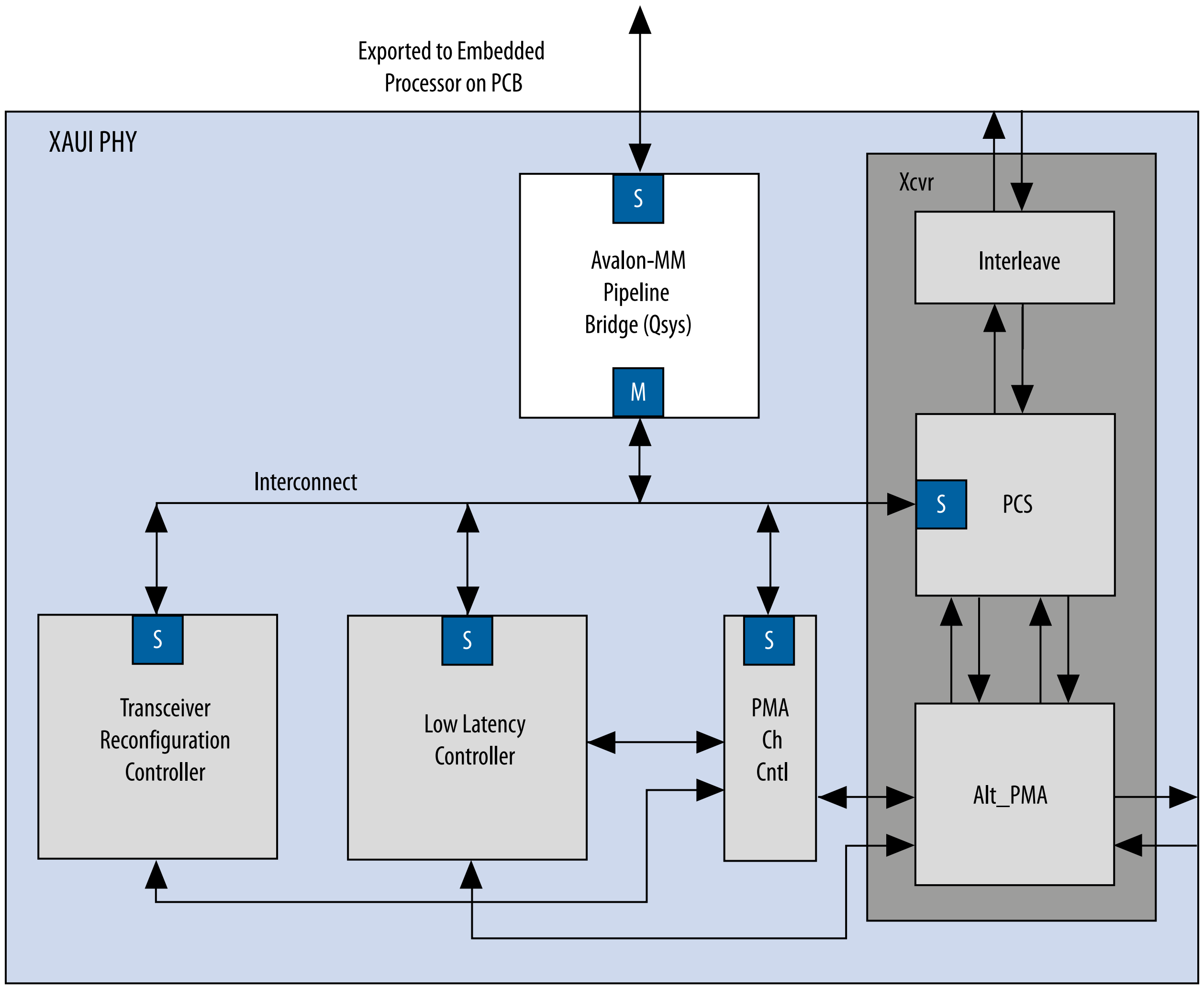
The Avalon-MM Pipeline Bridge inserts a register stage in the Avalon--MM command and response paths. The bridge accepts commands on its slave port and propagates the commands to its master port. The pipeline bridge provides separate parameters to turn on pipelining for command and response signals.

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 control multiple Avalon-MM slave devices. The pipelining feature is optional. You can optionally turn off the pipelining feature of this bridge.

Figure 9-4: 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.

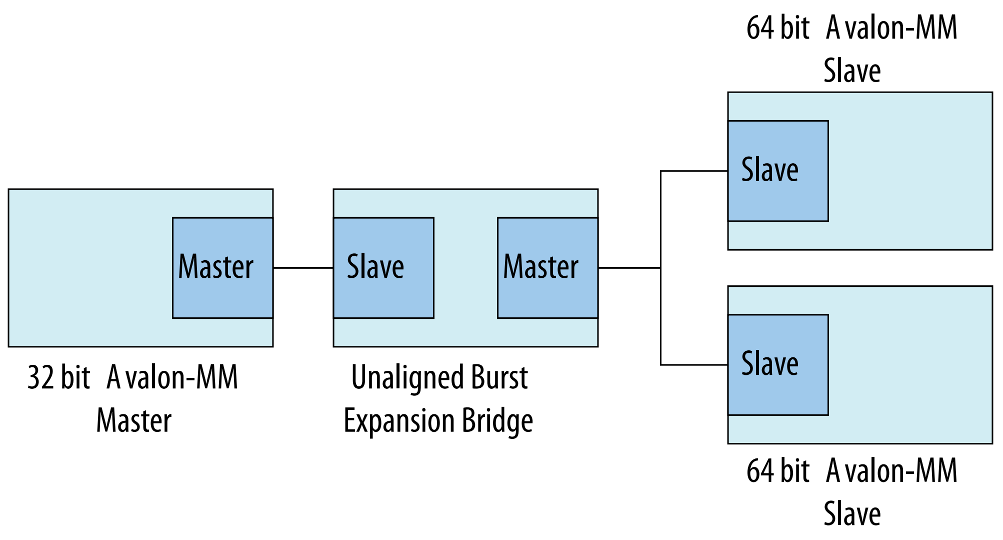

Exported to Embedded Processor on PCB
XAUI PHY
S
Avalon-MM Pipeline Bridge (Qsys)
M
Xcvr
Interleave
Interconnect
S
PCS
S
Transceiver Reconfiguration Controller
S
Low Latency Controller
S
PMA Ch Cntl
Alt_PMA

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 9-5: 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 unaligned 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: The Avalon-MM Unaligned Burst Expansion Bridge does not support VHDL simulation.

Related Information

- Qsys Interconnect on page 6-1

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.

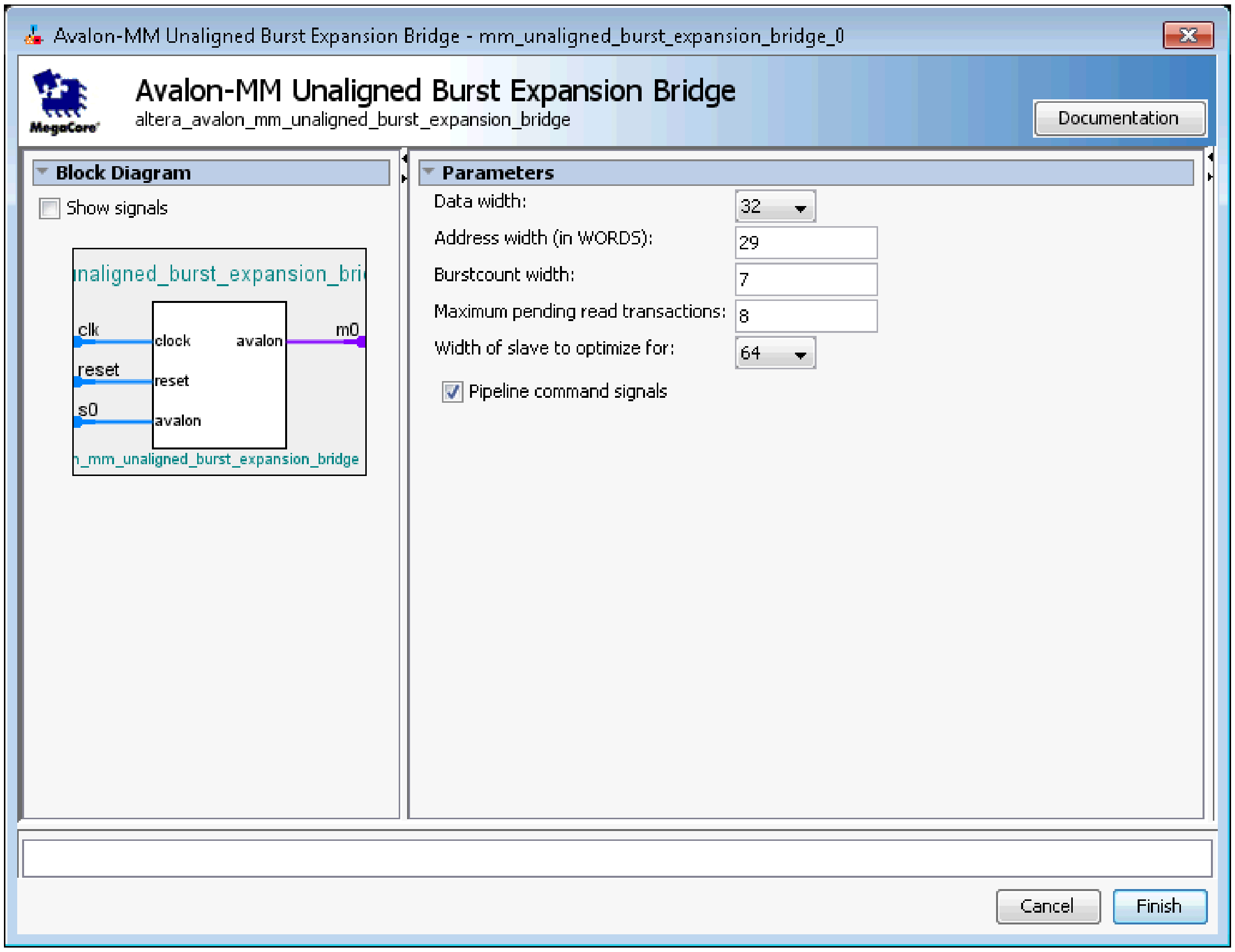
- If the final word requested by the master is the last word at the slave read burst address, the bridge does not modify the burst length of the read burst command to the slave.
- If the final word requested by the master is not the last word at the slave read burst address, the bridge increases the 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 9-6: 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
n_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 9-2: Avalon-MM Unaligned Burst Expansion Bridge Parameters

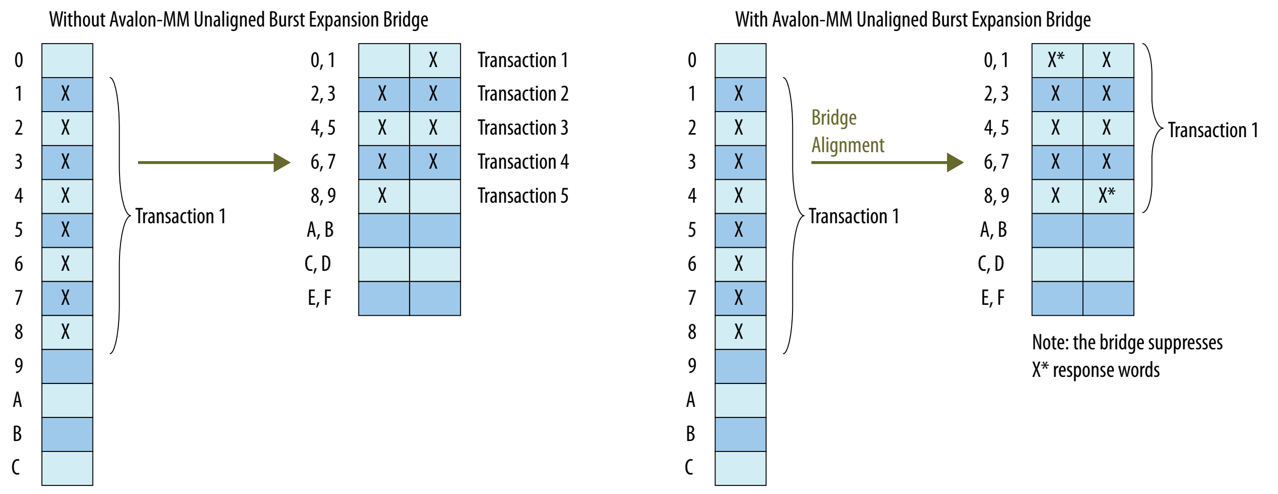
Parameter	Description
Data width	Data width of the master connected to the bridge.
Address width (in WORDS)	The address width of the master connected to the bridge.
Burstcount width	The burstcount signal width of the master connected to the bridge.

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 9-7: 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.



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.

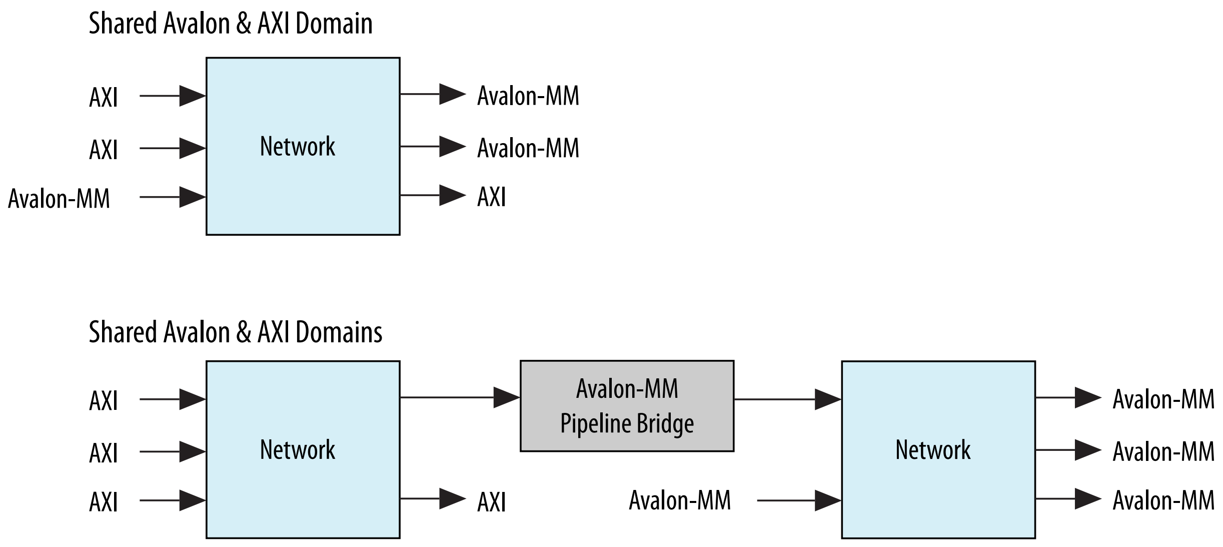
When the bridge receives the word corresponding to address 0, it suppresses it from the master, and then delivers the words corresponding to addresses 1 through 8 to the master. When the bridge receives the word corresponding to address 9, it suppresses that word from the master.

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 9-8: 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.



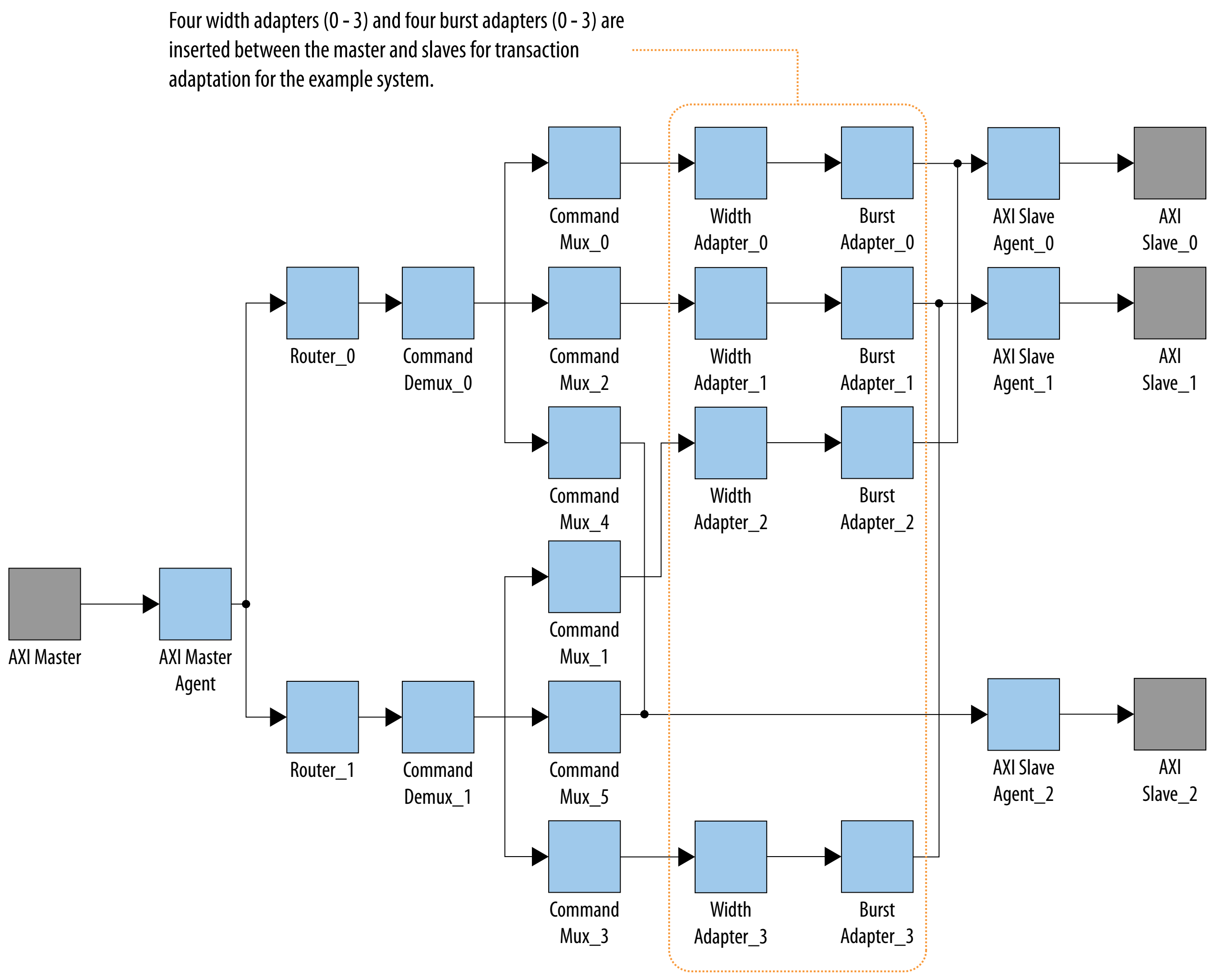
AXI Bridge

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.

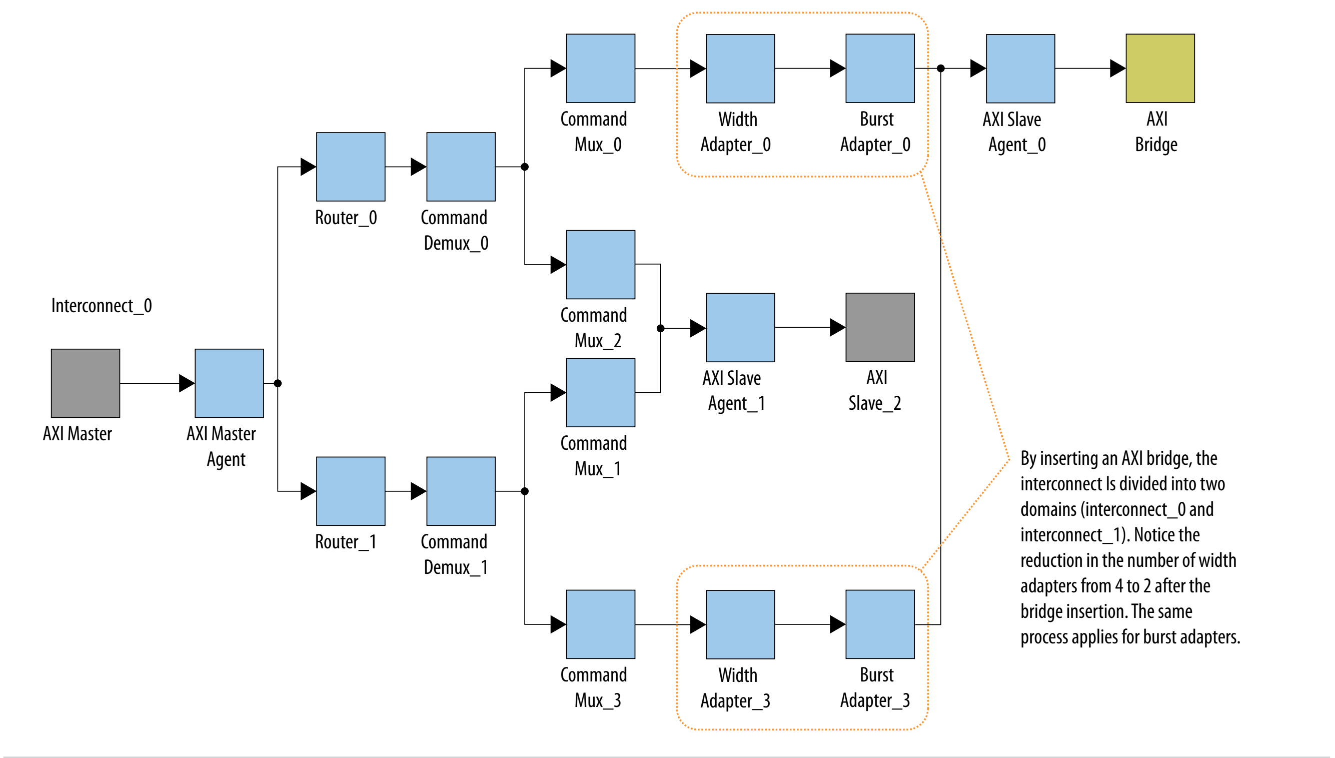
The example below shows a system with a single AXI master and three AXI slaves. It also has various interconnect components, such as routers, demultiplexers and multiplexer. Two of the slaves have a narrower data width than the master; 16-bit slaves versus a 32-bit master. In this system, Qsys intercon-nect creates four width adapters and four burst adapters to access the two slaves. You could improve resource usage by adding an AXI bridge. Then, Qsys needs to add only two width adapters and two burst adapters; one pair for the read channels, and another pair for the write channel.

Figure 9-9: AXI Example Without a Bridge: Adding a Bridge Can Reduce the Number of Adapters

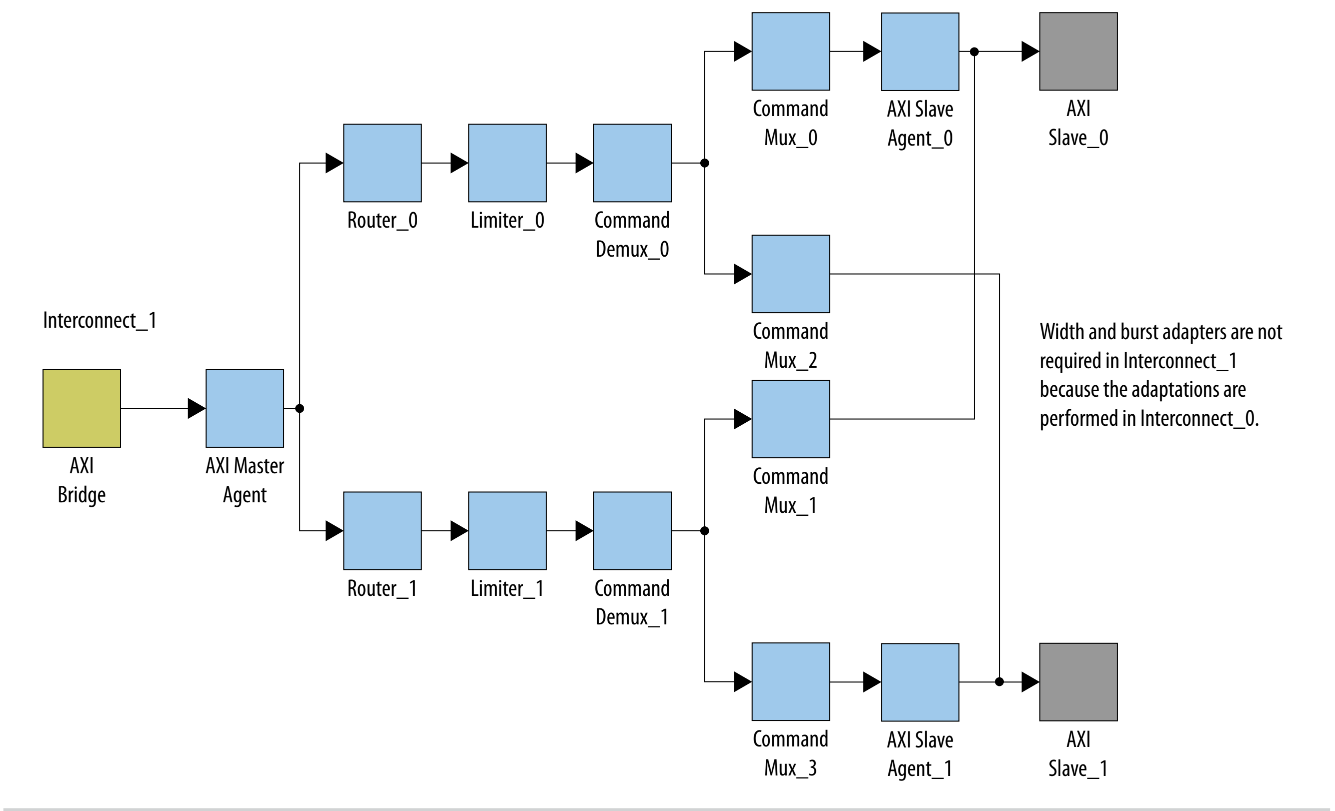


The example below shows the same system with an AXI bridge component, and the decrease in the number of width and burst adapters. Qsys creates only two width adapters and two burst adapters, as compared to the four width adapters and four burst adapters in the previous example. The system includes more components, but the overall system performance improves because there are fewer resource-intensive width and burst adapters.

Figure 9-10: Width and Burst Adapters Added to a System With a Bridge



By inserting an AXI bridge, the interconnect Is divided into two domains (interconnect_0 and interconnect_1). Notice the reduction in the number of width adapters from 4 to 2 after the bridge insertion. The same process applies for burst adapters.



Width and burst adapters are not required in Interconnect_1 because the adaptations are performed in Interconnect_0.

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).

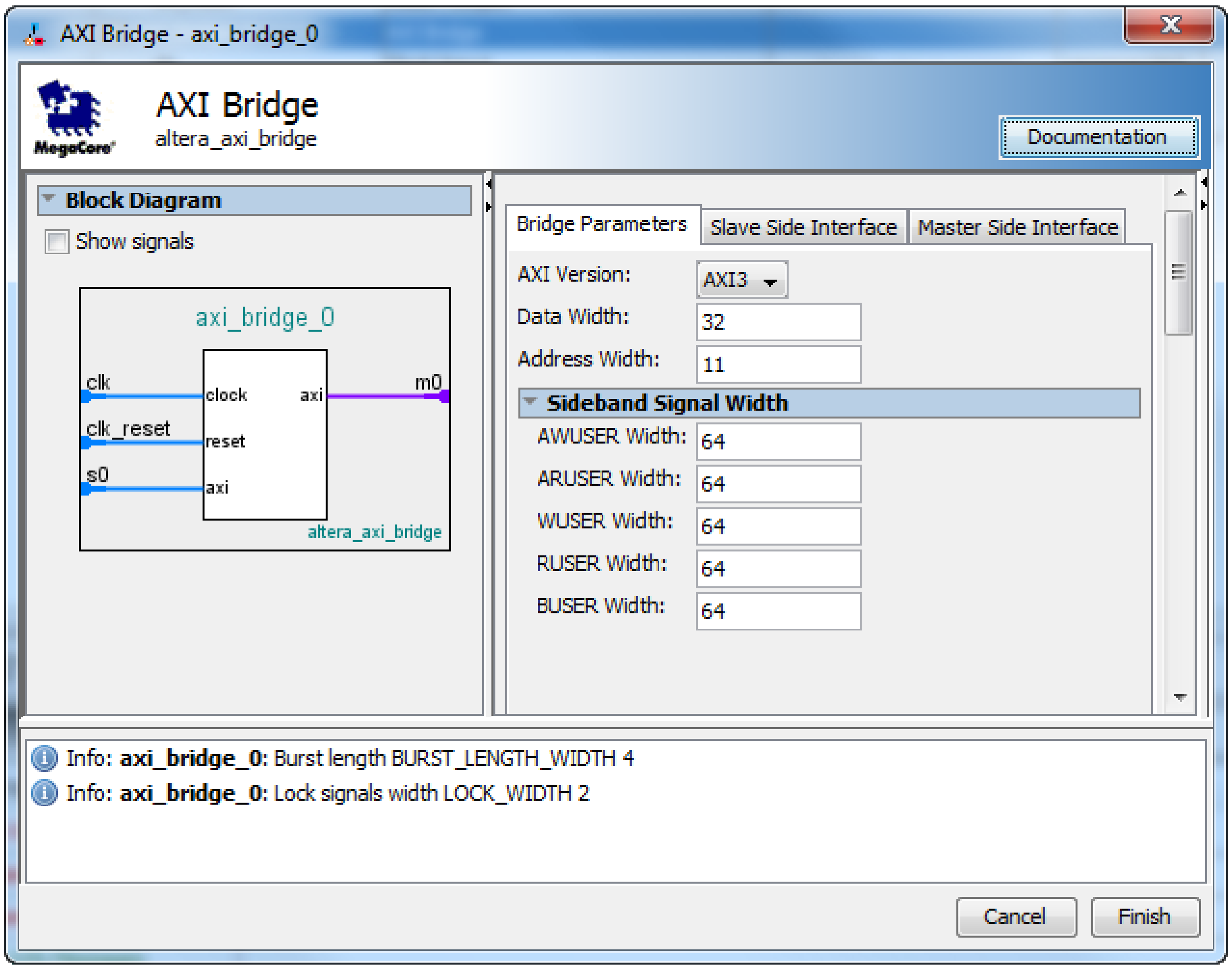
Table 9-3: Sets of Signals for the AXI Bridge Based on the Protocol

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 9-11: AXI Bridge Parameter Editor



AXI Version: AXI3
Data Width: 32
Address Width: 11
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 9-4: 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	integer	8:1024	Controls the width of the data for the master and slave interfaces.
Address Width	integer	1-64 bits	Controls the width of the address for the master and slave interfaces.

Parameter	Type	Range	Description
AWUSER Width	integer	1-64 bits	Controls the width of the write address channel sideband signals of the master and slave interfaces.
ARUSER Width	integer	1-64 bits	Controls the width of the read address channel sideband signals of the master and slave interfaces.
WUSER Width	integer	1-64 bits	Controls the width of the write data channel sideband signals of the master and slave interfaces.
RUSER Width	integer	1-16 bits	Controls the width of the read data channel sideband signals of the master and slave interfaces.
BUSER Width	integer	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 9-5: AXI Bridge Slave and Master Interface Parameters

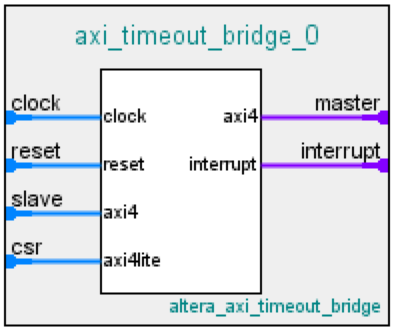
Parameter	Description
ID Width	Controls the width of the thread ID of the master and slave interfaces.
Write/Read/Combined Acceptance Capability	Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the slave interface accepts.
Write/Read/Combined 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 time out 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 also facilitates debugging.

Figure 9-12: AXI Timeout Bridge

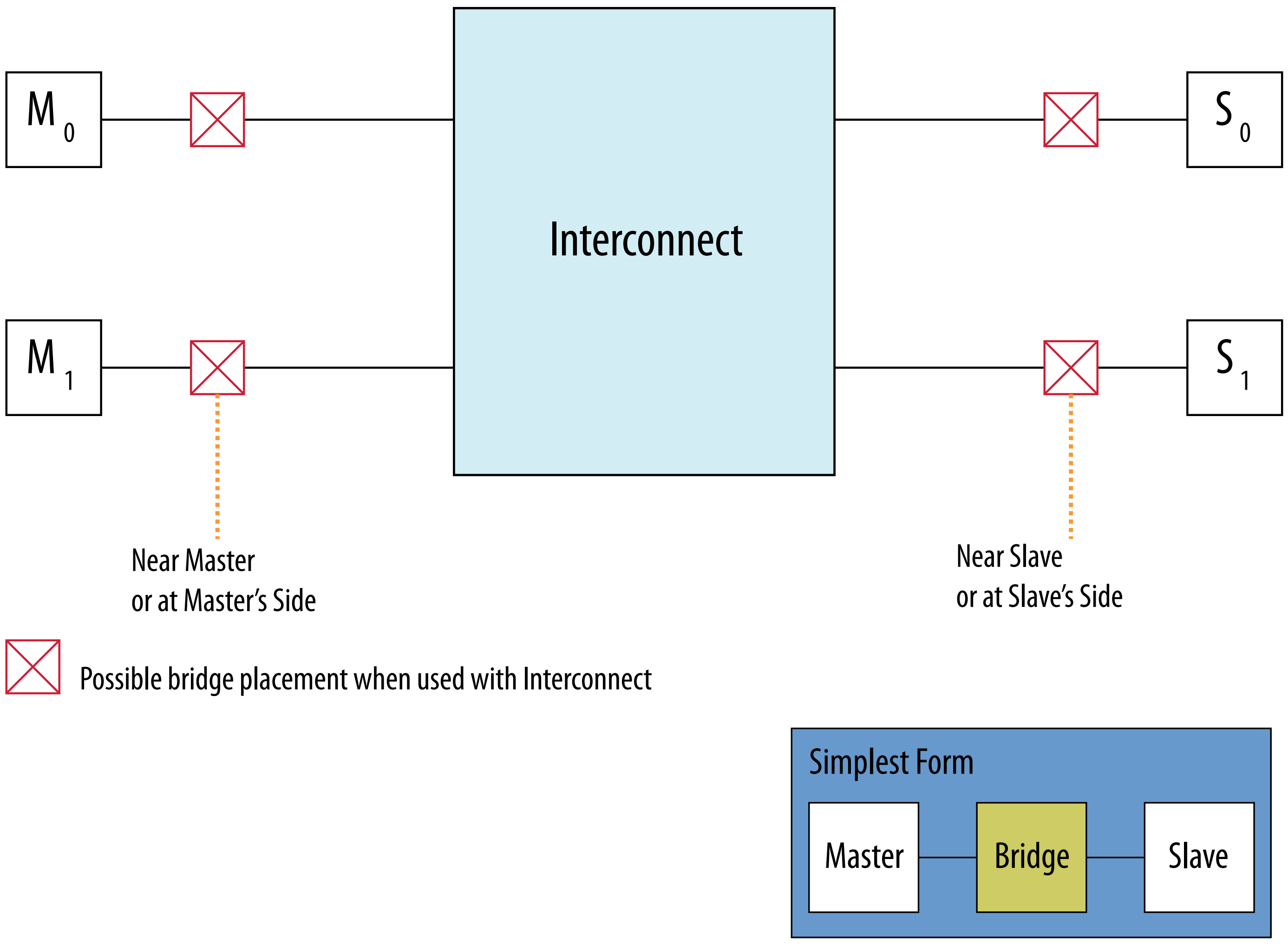


For a domain with multiple masters and slaves, placement of an AXI Timeout Bridge in your design may be beneficial in the following scenarios:

- 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.

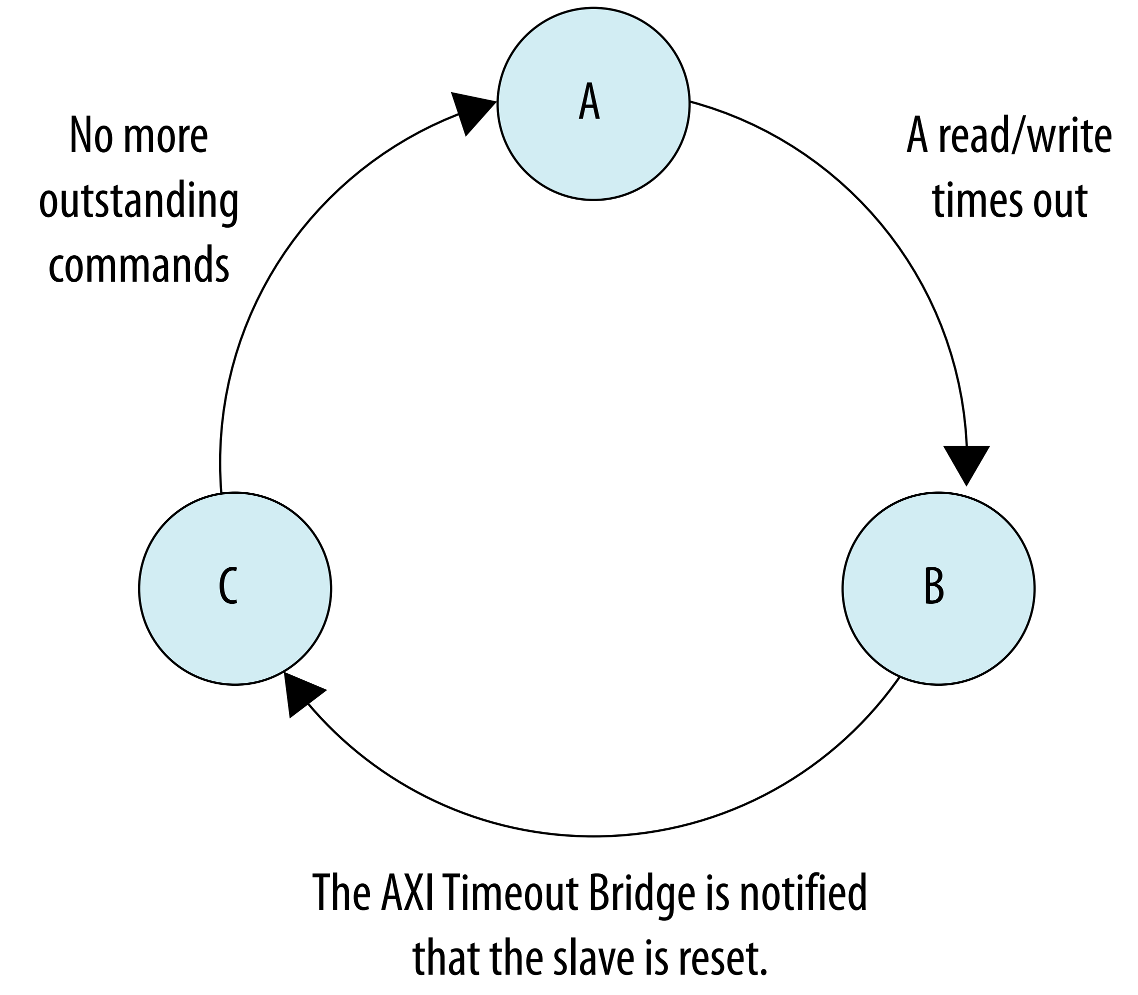
Figure 9-13: AXI Timeout Bridge Placement



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 9-14: 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 Configuration and Status Register (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 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 9-6: 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 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 9-7: 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 9-8: 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.

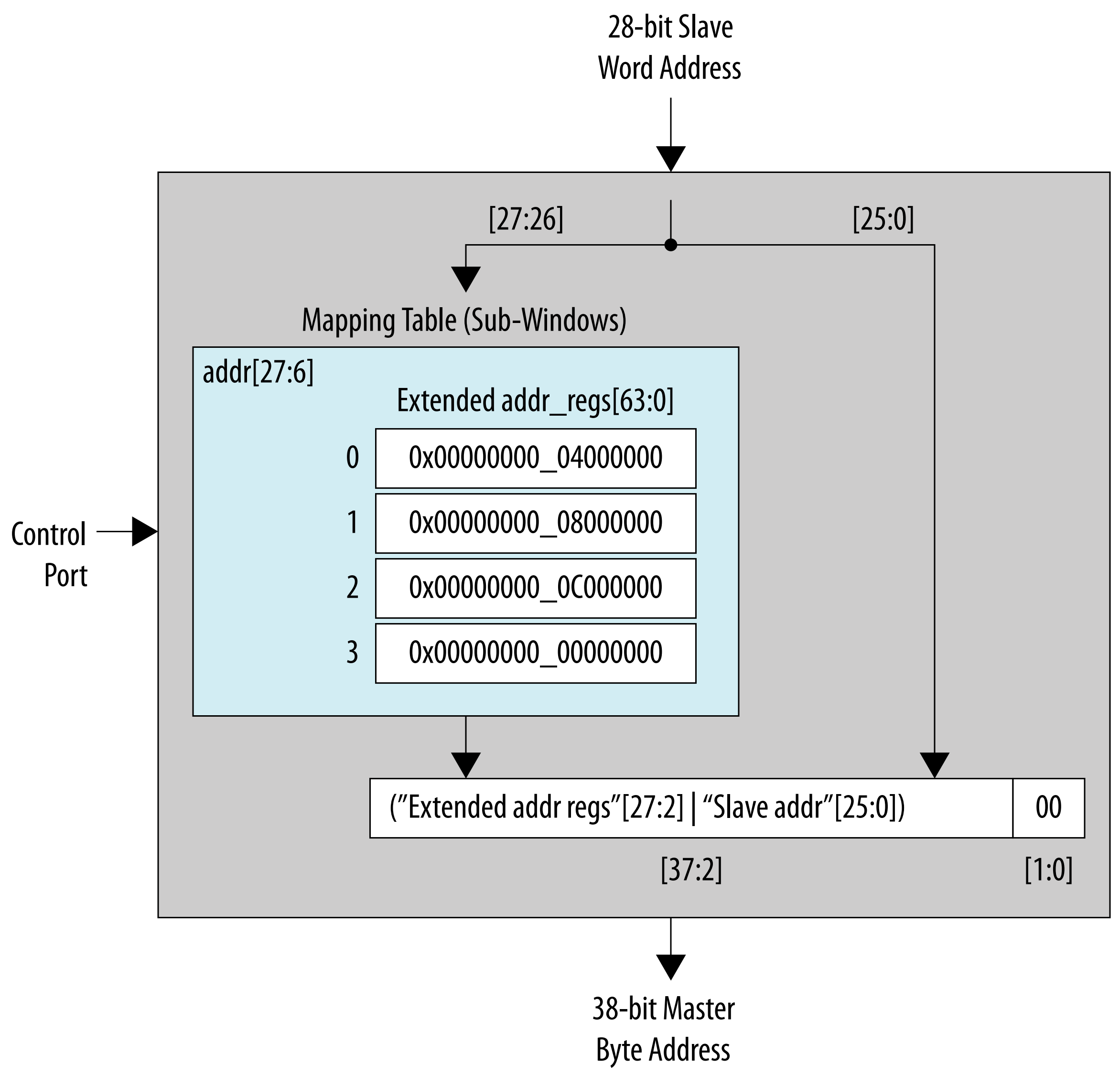
CTRL Register Layout

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. The lower 26 bits originate from the address span extender's data port. The upper 2 bits `[27:26]` originate from the control registers.

Figure 9-15: 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.

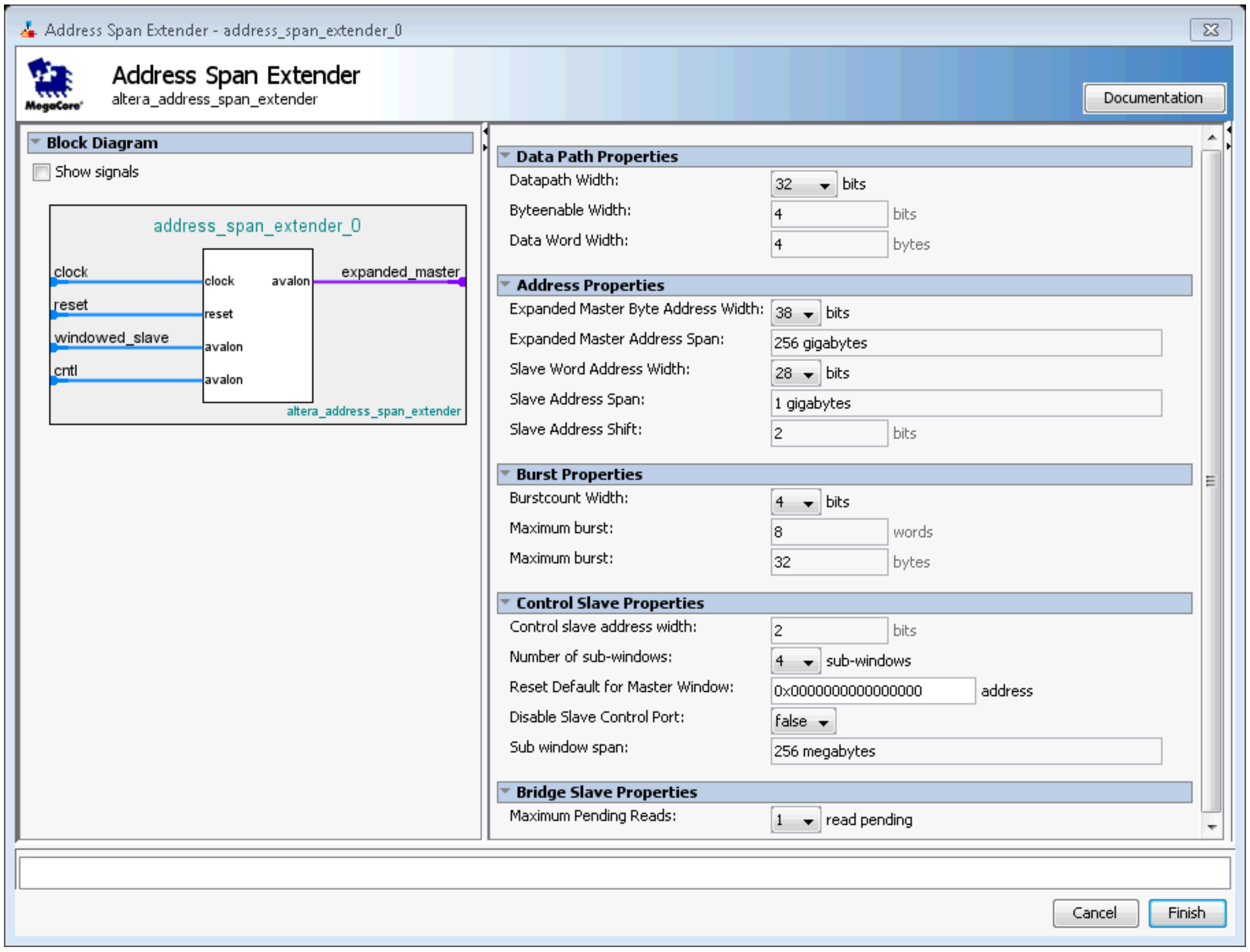
Each sub-window has a 64-bit register set that defines the sub window's upper address, and use only the bits greater than the slave byte address.

- **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 9-16: 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

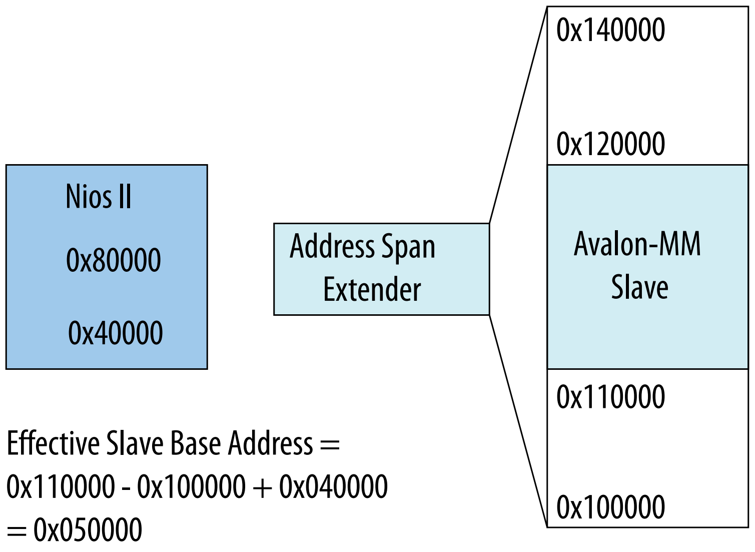
If the address span extender window is fixed, for example, the **Disable Slave Control Port** option is turned on, then the address span extender performs as a bridge. Components on the slave side of the address span extender that are within the window are visible to the NIOS II processor. Components 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 9-17: 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.

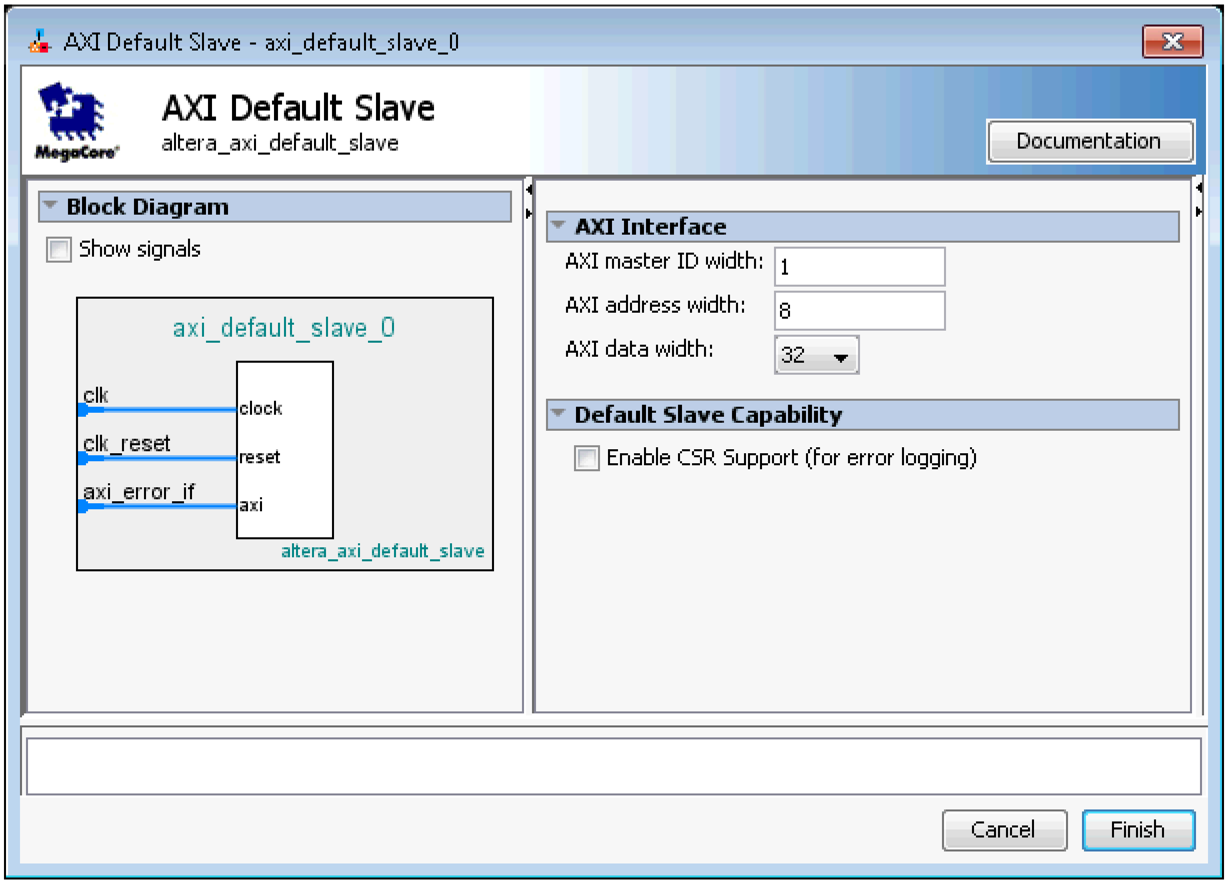
For information about creating secure systems and accessing undefined memory regions, refer to *Creating a System with Qsys* in volume 1 of the *Quartus Prime Handbook*.

Related Information

- **Creating a System with Qsys** on page 4-1

AXI Default Slave Parameters

Figure 9-18: AXI Default Slave Parameter Editor



AXI Interface

AXI master ID width: 1

AXI address width: 8

AXI data width: 32

Default Slave Capability

Enable CSR Support (for error logging)

Table 9-9: 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 9-10: CSR Interrupt Status Registers

For CSR register maps: `Address = Memory Address Base + Offset`.

Offset	Bit	Attribute	Default	Descripton
`0x00`	`31:4`	RO	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 9-11: 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 9-12: 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	RO	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	RO	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	RO	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	RO	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 9-19: 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.

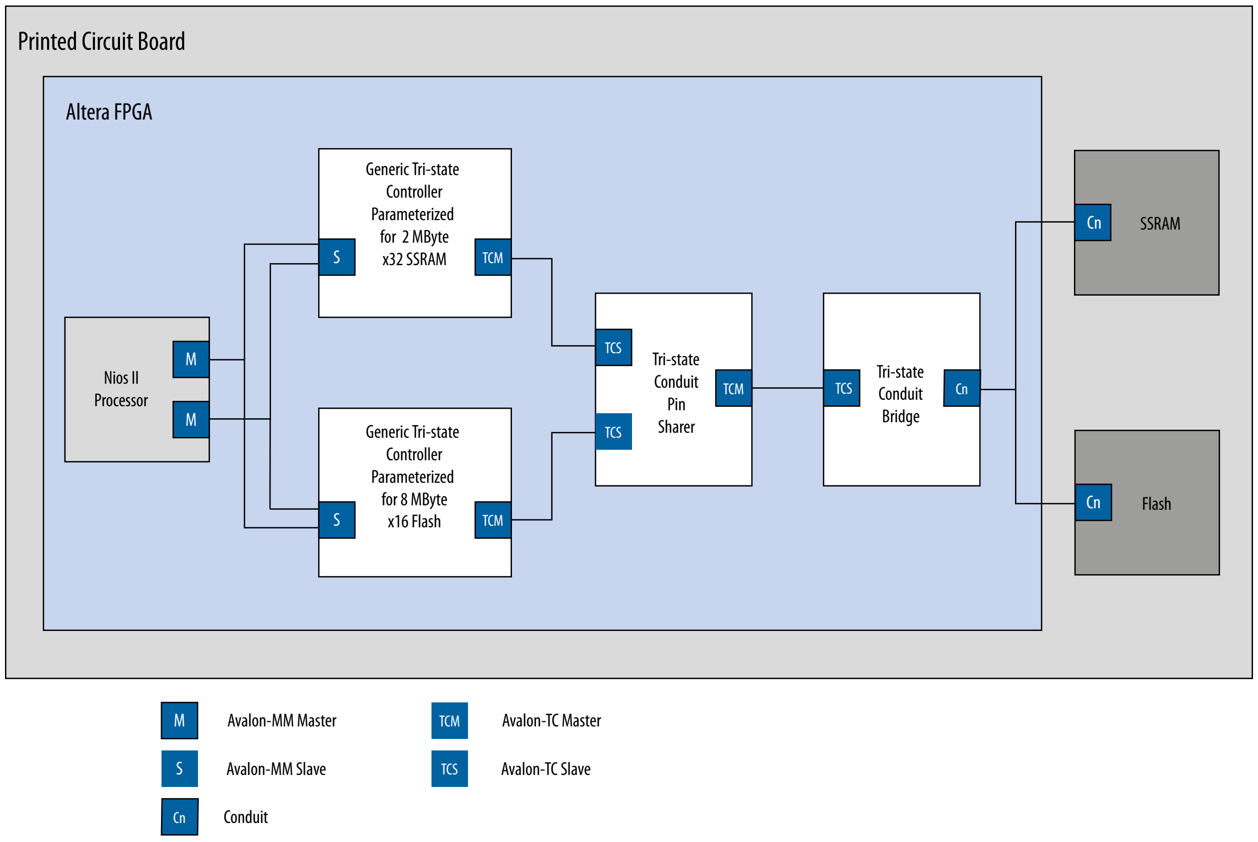
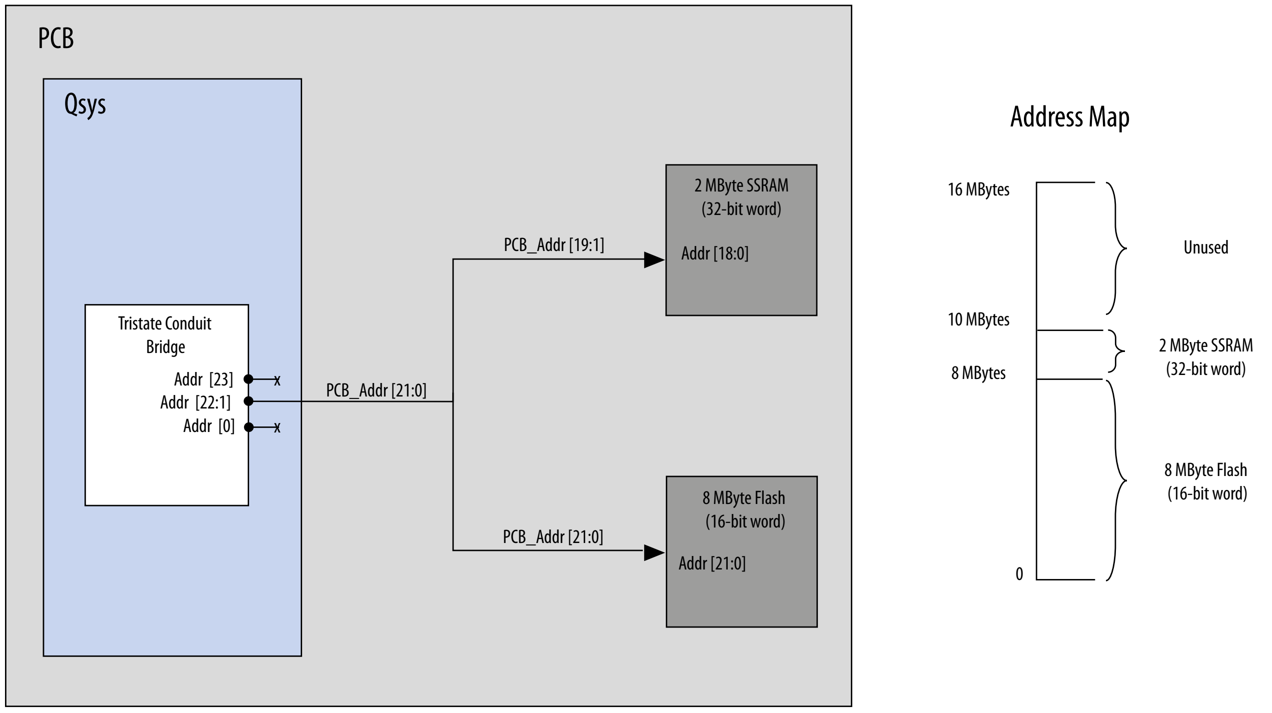


Figure 9-20: Address Connections from Qsys System to PCB

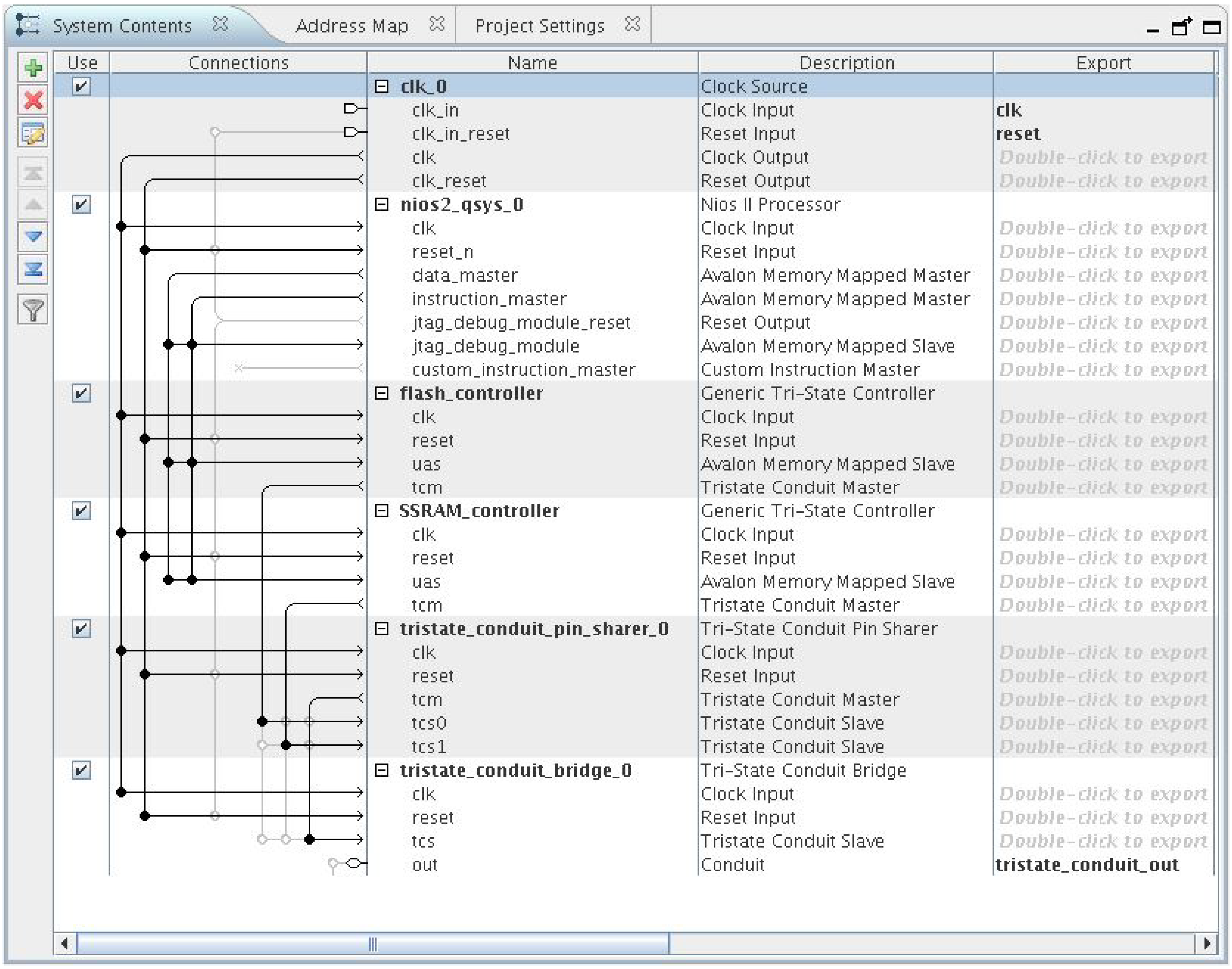
The flash device operates on 16-bit words and must ignore the least-significant bit of the Avalon-MM address, and shows `addr[0]`as not connected. The SSRAM memory operates on 32-bit words and must ignore the two, low-order memory bits. Because neither device requires a byte address, `addr[0]` is not 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 9-21: 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:

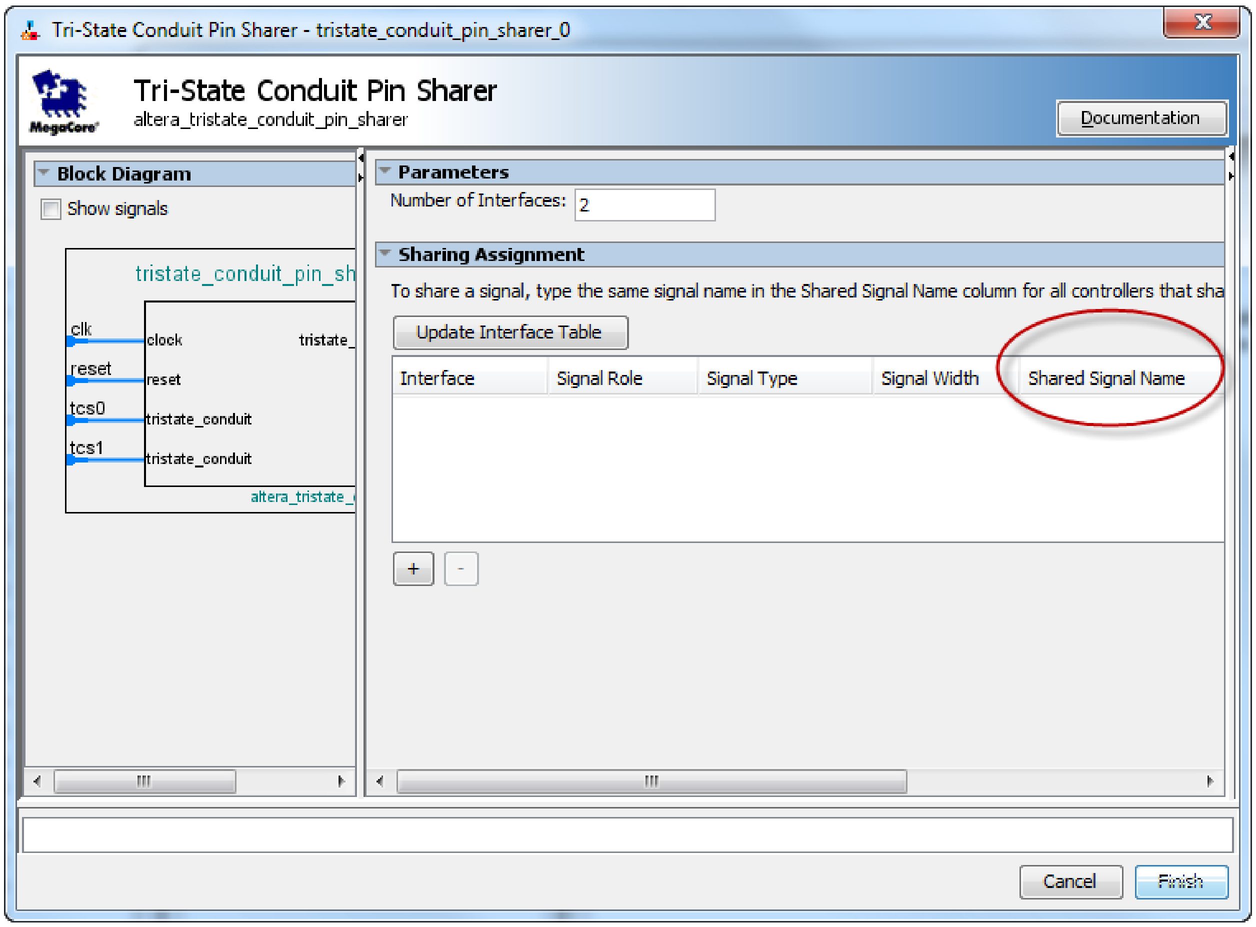
- **Memory-mapped slave interface**—This interface connects to an memory-mapped master, such as a processor.
- **Tristate Conduit Master interface**—Tri-state master interface usually connects to the tri-state conduit slave interface of the tri-state conduit pin sharer.
- **Clock sink**—The component's 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 9-22: 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 9-66

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

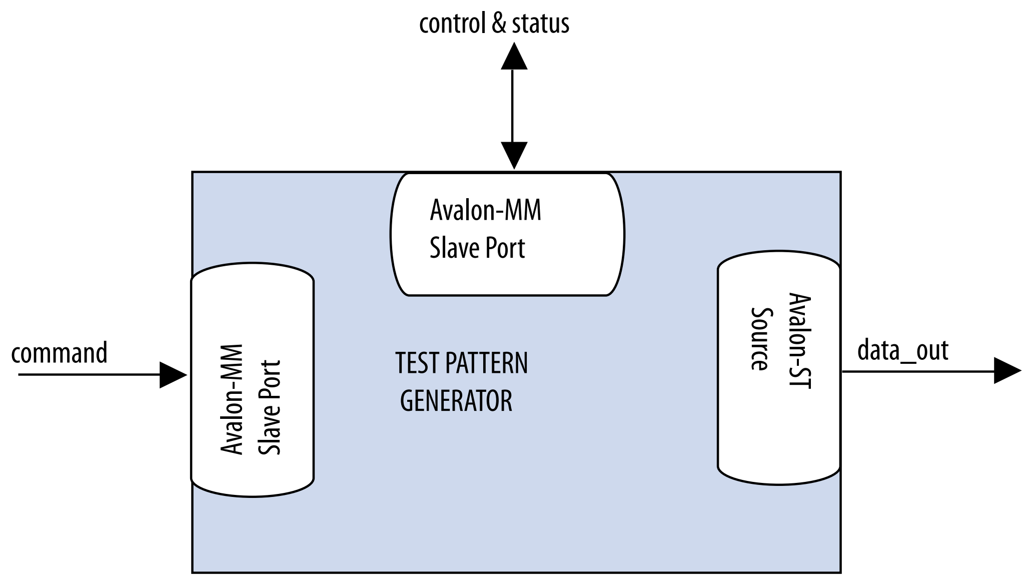
The data generation and monitoring solution for Avalon-ST consists of two components: a test pattern generator core that generates data, and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and verifies it. Optionally, the data can be formatted 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 9-23: 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.

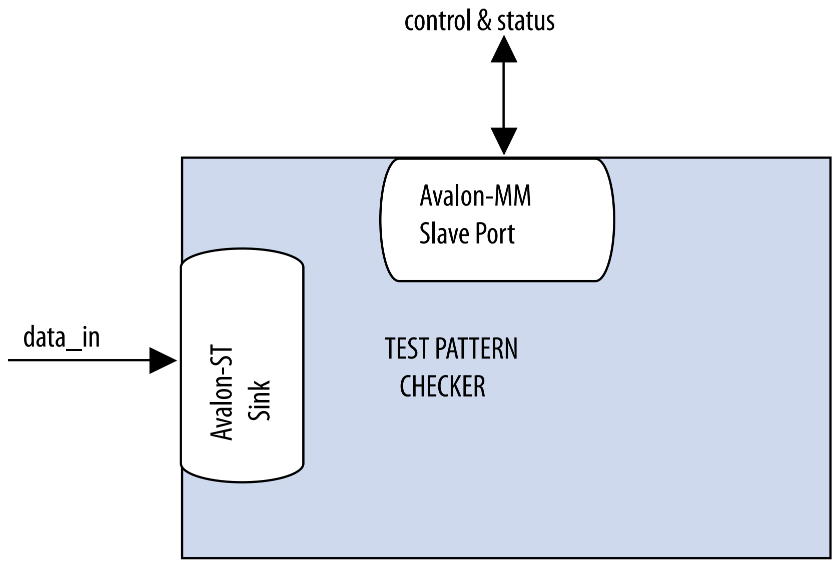
Test Pattern Generator Functional Parameter

The Test Pattern Generator functional parameter allows you to configure the test pattern generator as a whole system.

Test Pattern Checker

Figure 9-24: 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 9-13: 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 9-14: 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 9-15: 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 9-16: 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 9-17: 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 9-18: `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 9-19: `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 9-20: 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 9-21: 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 9-22: 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 9-23: `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 9-24: `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 9-25: `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 9-26: 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 9-27: 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 9-28: 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 9-29: 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 9-30: 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 9-31: 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 9-32: 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 9-33: 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 9-34: 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 9-35: 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 9-36: 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 9-37: 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 9-38: 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 9-53

data_sink_init() on page 9-53

data_sink_get_id() on page 9-54

data_sink_get_supports_packets() on page 9-54

data_sink_get_num_channels() on page 9-55

data_sink_get_symbols_per_cycle() on page 9-55

data_sink_get_enable() on page 9-55

data_sink_set enable() on page 9-56

data_sink_get_throttle() on page 9-56

data_sink_set_throttle() on page 9-57

data_sink_reset()

Table 9-39: 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 9-40: 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 9-41: 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 9-42: 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 9-43: 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 9-44: 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 9-45: 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 9-46: 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 9-47: 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 9-48: 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 9-49: 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 9-50: 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 9-51: 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 9-52: 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 9-53: 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 9-54: 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 9-55: 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 9-56: 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 9-57: 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()

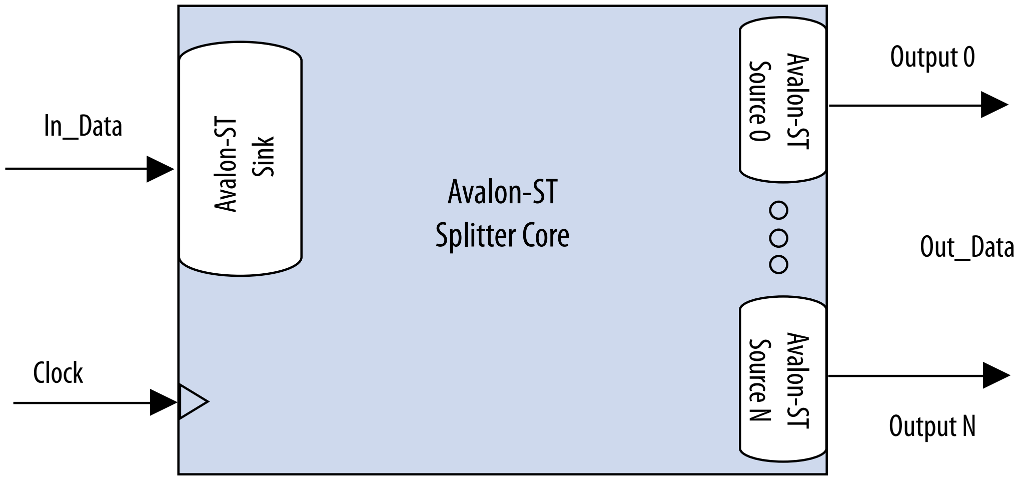
Table 9-58: data_sink_exception_channel()

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 9-25: 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 9-59: 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 9-60: Avalon-ST Splitter Core Parameters

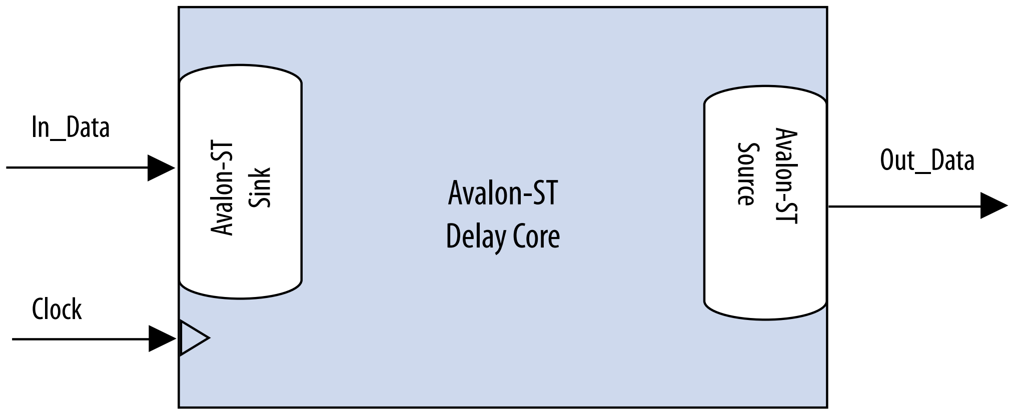
Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. Qsys supports 1 for some systems where no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `out_valid` signal is gated or non-gated when backpressure is applied.
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 9-26: 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 9-61: 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 9-62: Avalon-ST Delay Core Parameters

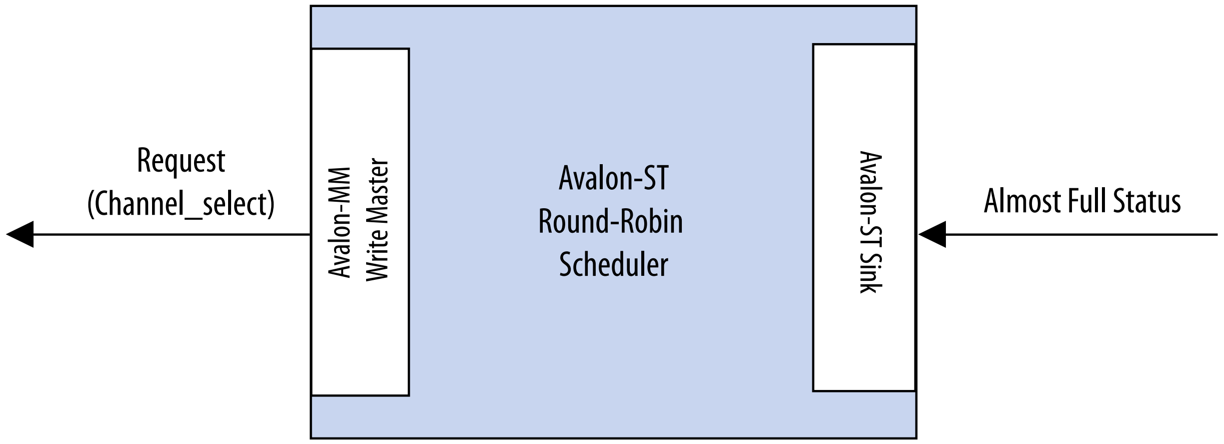
Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. Qsys supports 0 for some systems where no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The 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 9-27: 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 9-63: 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 9-64: 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

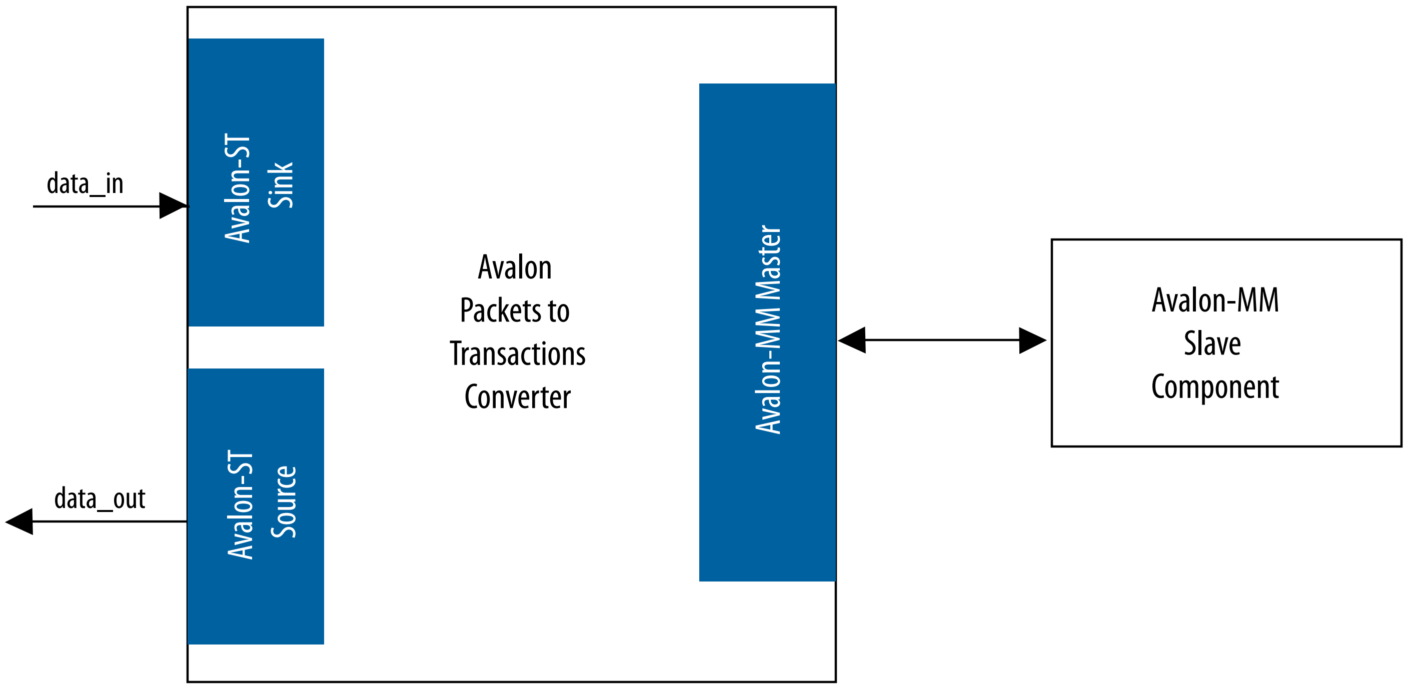
Table 9-65: Avalon-ST Round Robin Scheduler Parameters

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 9-28: 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
data_out
Avalon-ST Sink
Avalon-ST Source
Avalon Packets to Transactions Converter
Avalon-MM Master
Avalon-MM Slave Component

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 9-66: 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 9-67: 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 9-69
- **Packets to Transactions Converter Interfaces** on page 9-69

Packets to Transactions Converter Supported Transactions

Table 9-68: 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 interfa-ce'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 9-29: Pipeline Stage Simple Register

If the ready signal is not pipelined, the pipeline stage becomes a simple register.

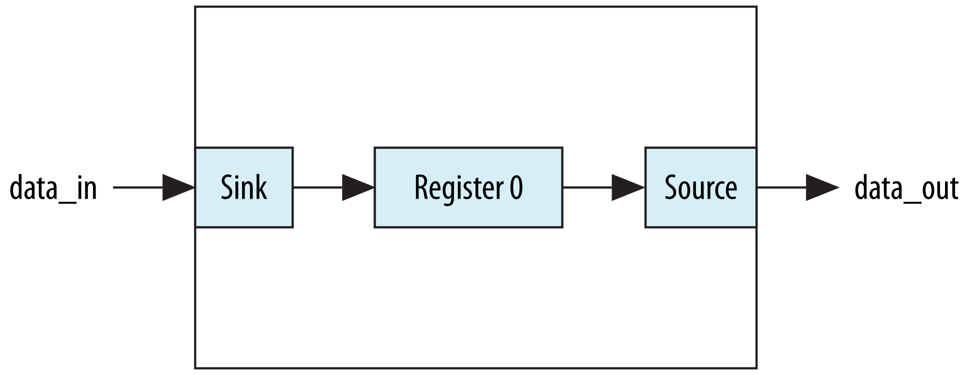
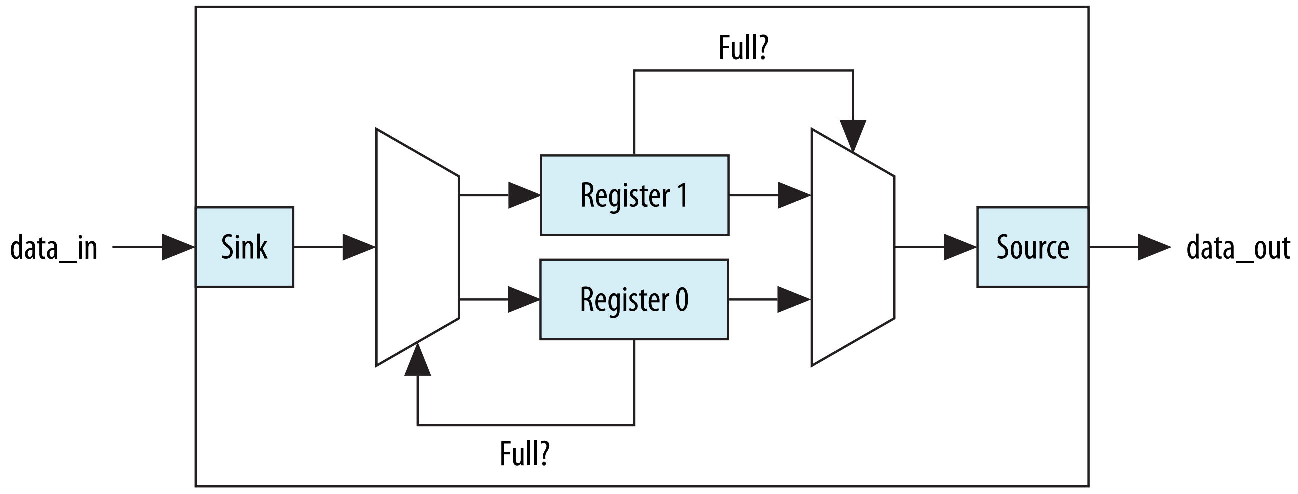


Figure 9-30: Pipeline Stage Holding Register

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 9-66

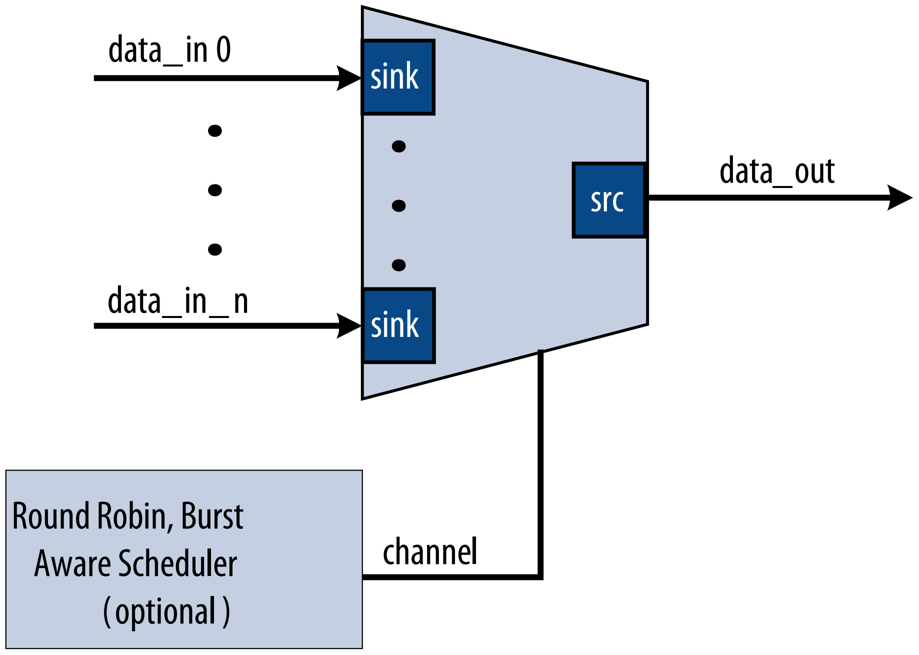
Software Programming Model For the Multiplexer and Demultiplexer Components

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 9-31: 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

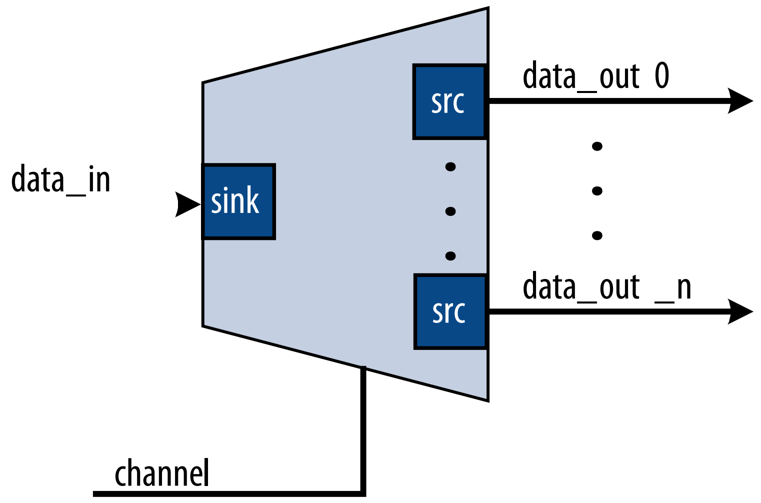
You can configure the following parameters for the multiplexer:

- **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 9-32: 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.

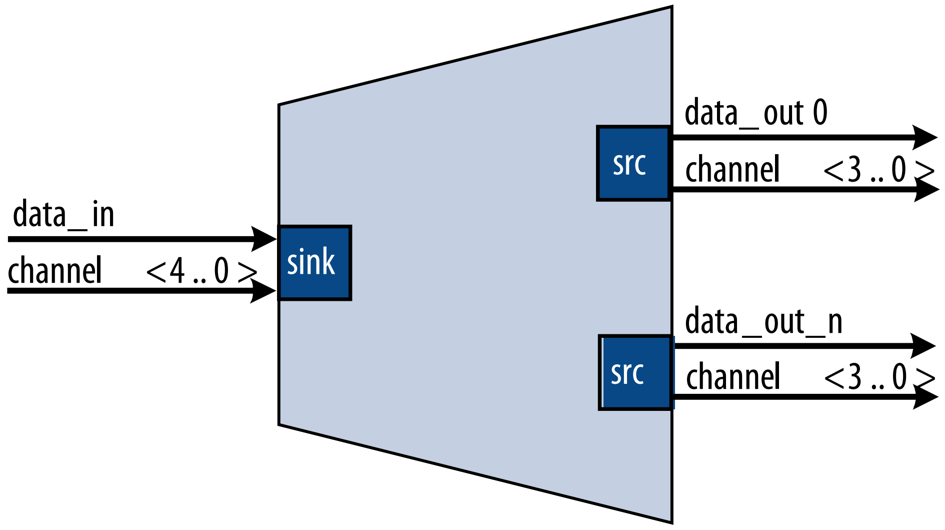
Demultiplexer Parameters

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 9-33: 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 9-34: Avalon-ST Single Clock FIFO Core

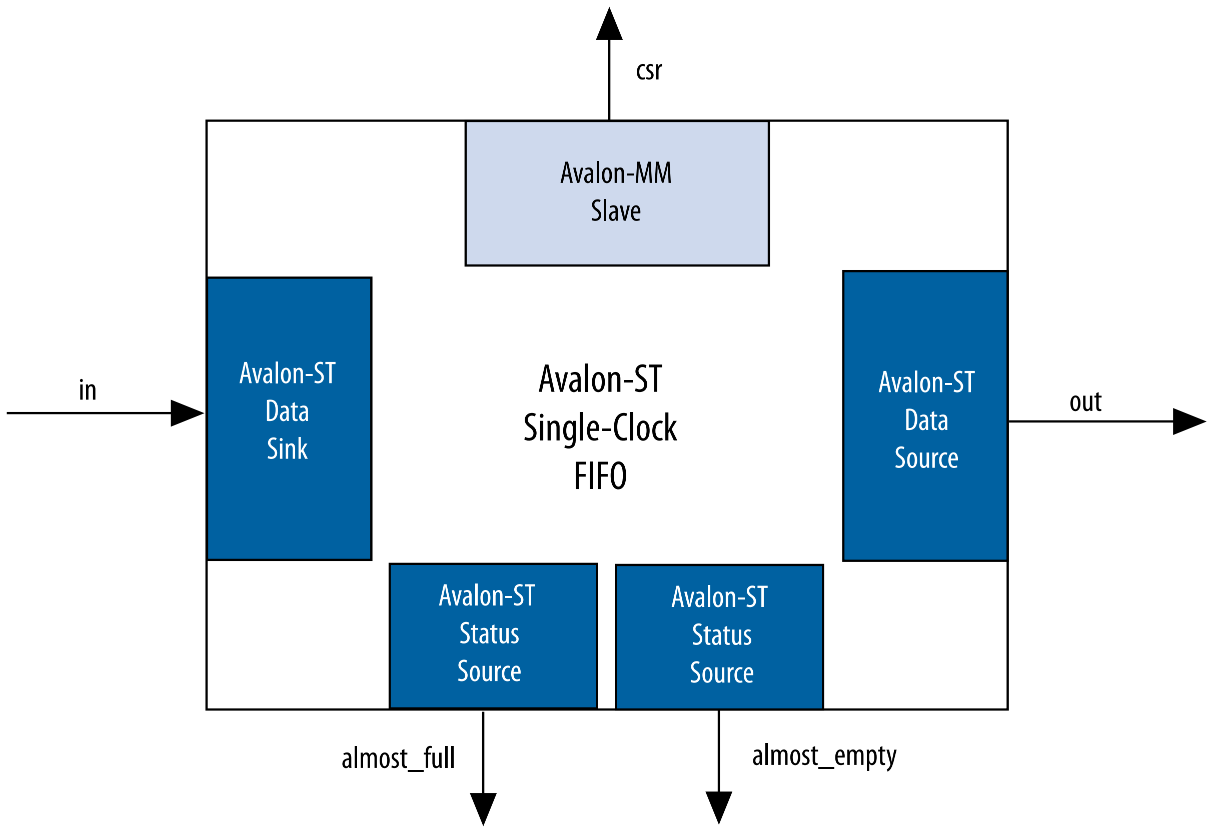
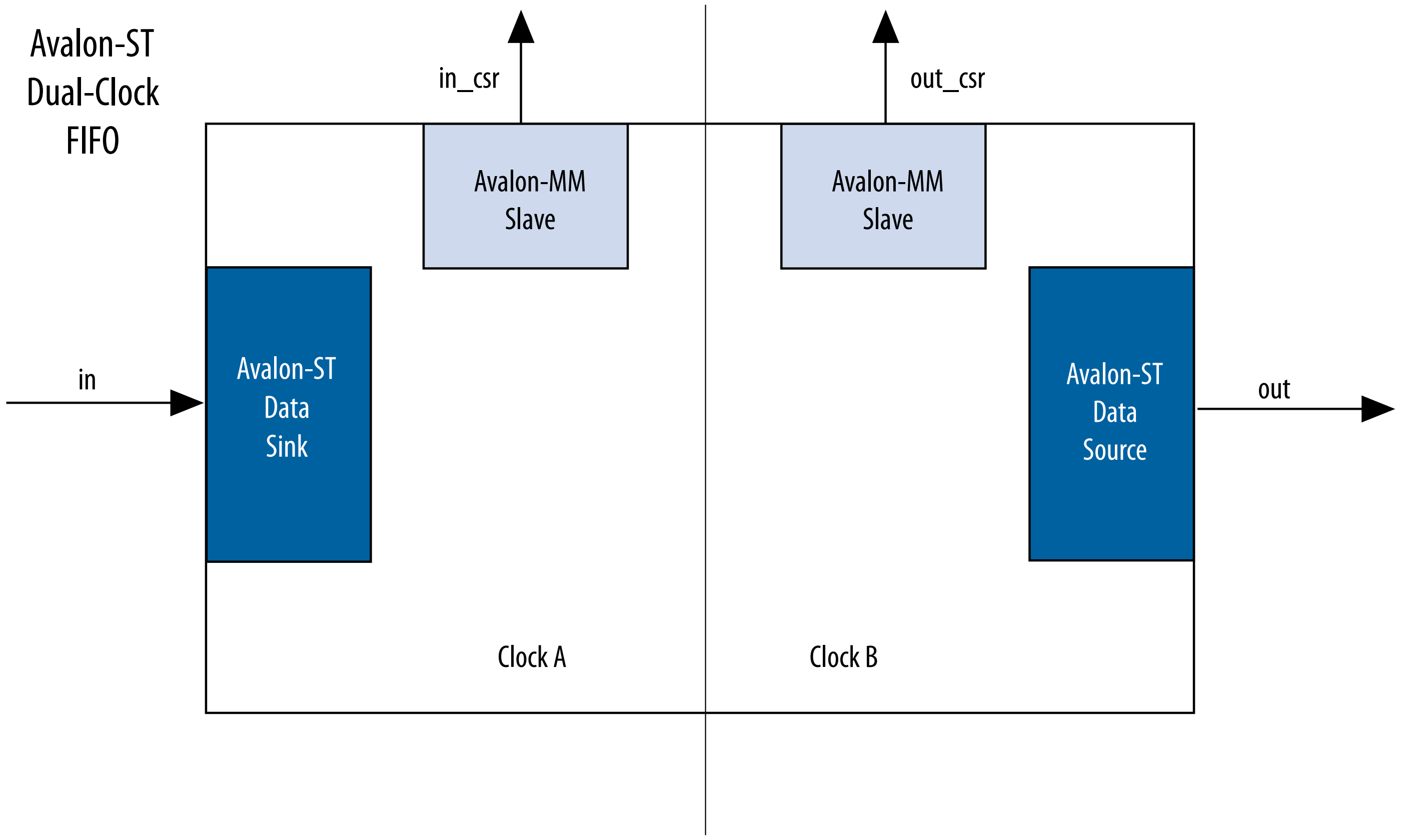


Figure 9-35: Avalon-ST Dual Clock FIFO Core



Interfaces Implemented in FIFO Cores

The following interfaces are implemented in FIFO cores:

Avalon-ST Data Interface on page 9-79

Avalon-MM Control and Status Register Interface on page 9-80

Avalon-ST Status Interface on page 9-80

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 9-69: 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 9-70: 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. 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.
Symbols per beat	1–32	
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 9-71: 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 9-72: 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 9-73: Document Revision History

The table below indicates edits made to the *Qsys System Design Components* content since its creation.

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
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 Handbook Archive

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

Recommended Design Practices 10

2015.11.02

QPP5V1 Subscribe Send Feedback

This chapter provides design recommendations for Altera® devices and describes the Quartus Prime 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 Prime 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.

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).

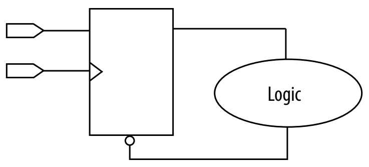
For cases where combinational logic feeds registers, the register control signals can implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

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 combina-tional logic.

Figure 10-1: 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 Prime 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.

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 Prime Text Editor or Block Editor.

It is common for mistakes in HDL code to cause unintended latch inference; Quartus Prime 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.

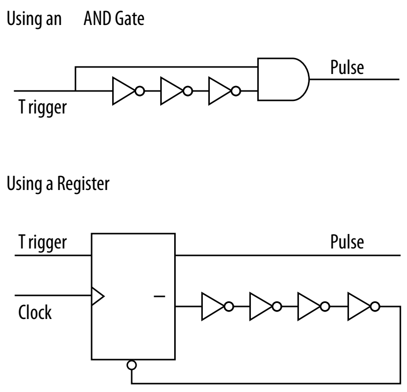
Delays in PLD designs can change with each placement and routing cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Avoid using delay chains to prevent these kinds of 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 10-2: 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.

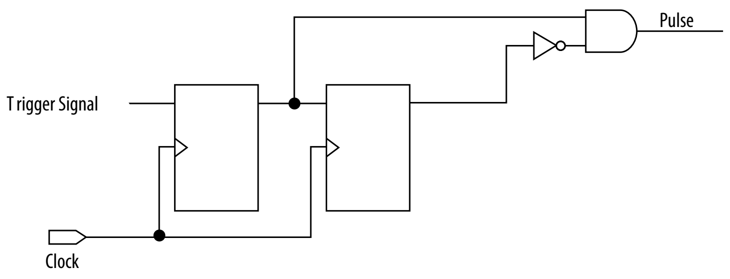
A register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

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 10-3: 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 Prime 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 Prime 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.

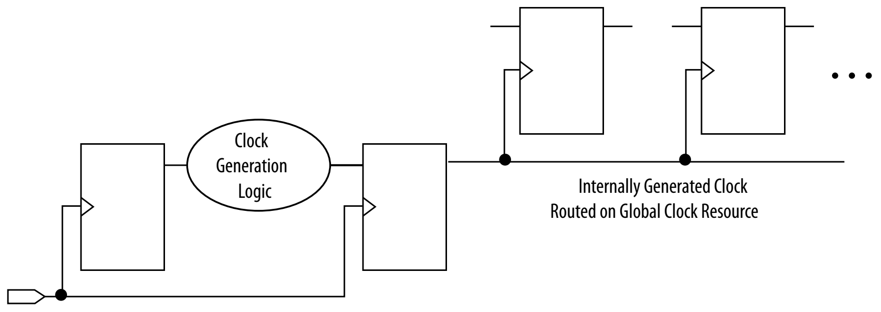
Register Combinational Logic Outputs

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 10-4: 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.

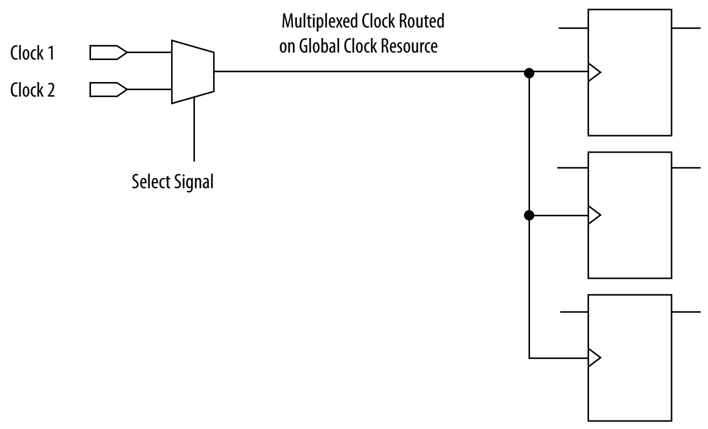
You can often use ripple clock structures to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus Prime software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows 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 10-5: 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 Prime 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 Prime software, so that all register-to-register paths are analyzed using that clock.

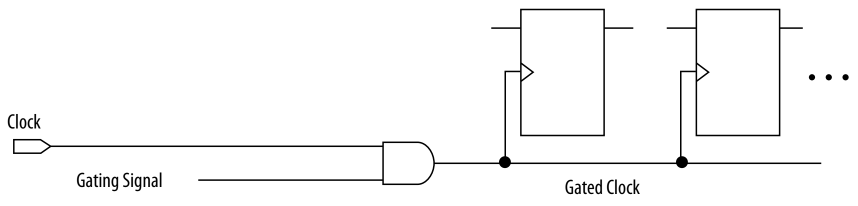
Tip: Use dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the clock-switchover feature or clock control block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing 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 10-6: 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.

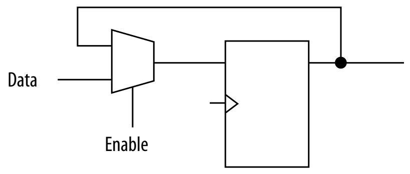
From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the 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 10-7: Synchronous Clock Enable



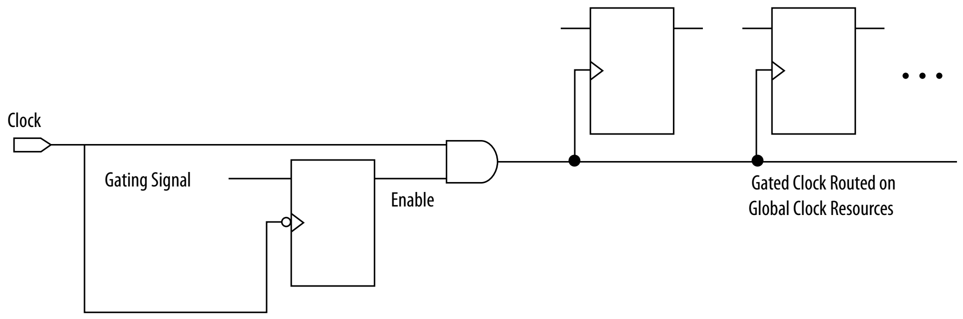
Recommended Clock-Gating Methods

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 10-8: 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 Prime 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 Prime 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 Prime 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.

Checking Design Violations

To improve the reliability, timing performance, and logic utilization of your design, avoid design rule violations. The Quartus Prime 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.

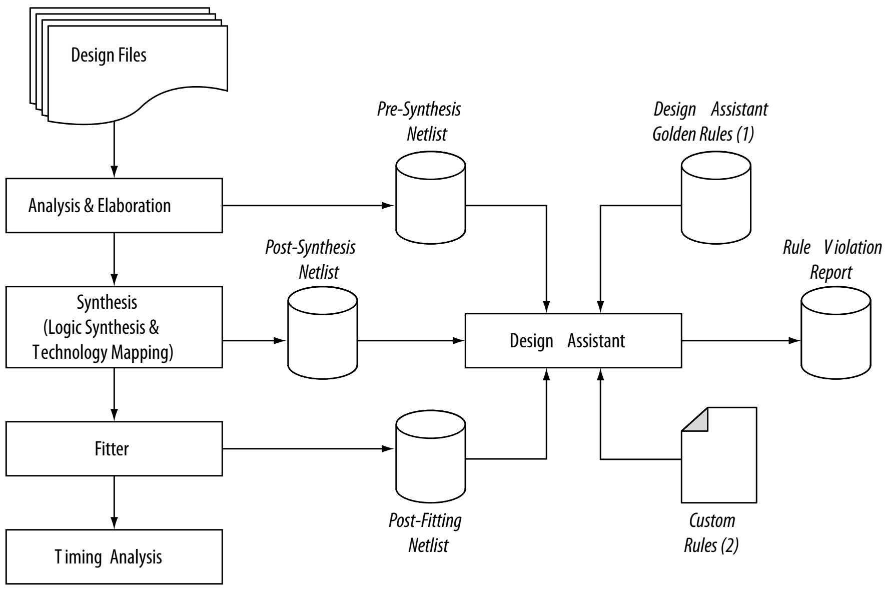
This section provides an introduction to the Quartus Prime design flow with the Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last parts of the section describe the design rules and the reports generated by the Design Assistant. The Design Assistant supports all Altera devices supported by the Quartus Prime 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 10-9: Quartus Prime 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 Prime 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

Design Assistant Rules

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 Prime 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 10-1: 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>
```

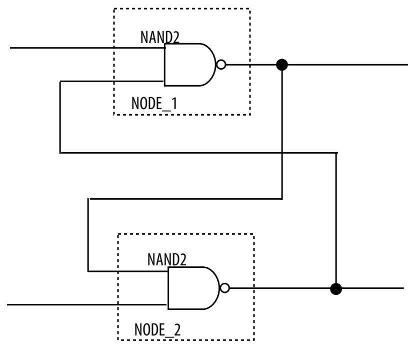
The possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the 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 10-2: Detecting SR Latches in a Design

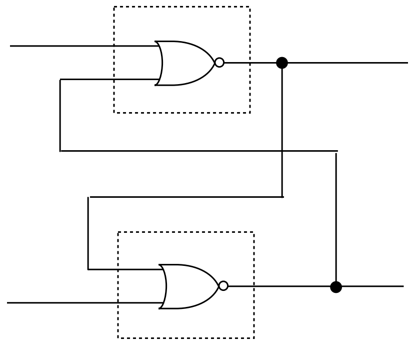
```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
   TO_TYPE="NAND" />
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
   TO_TYPE="NAND" />
</AND>
```

Figure 10-10: 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 10-11: 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 10-3: 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>`.

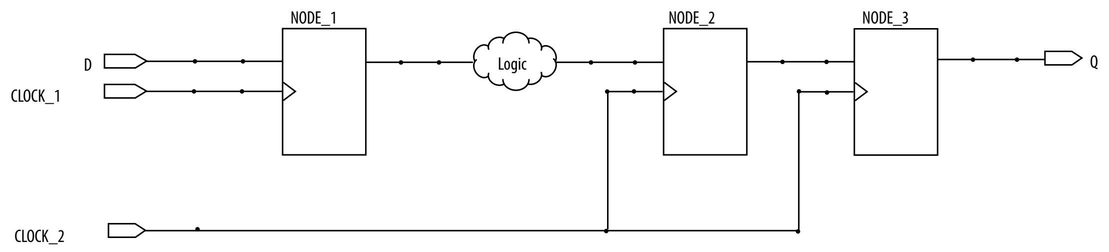
The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the SR latch structures. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples show the undesired conditions from with their equivalent block diagrams:

Example 10-4: 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" />
```

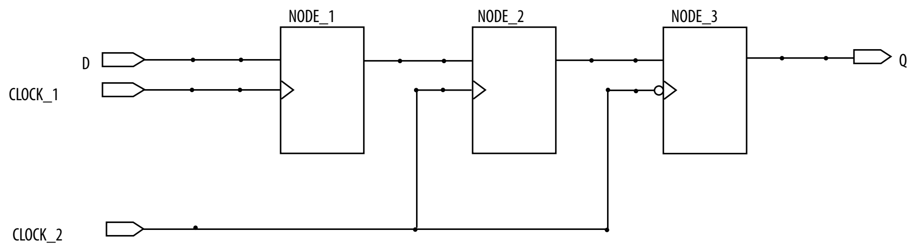
Figure 10-12: Undesired Condition 3



Example 10-5: 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 10-13: 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 Prime 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 Prime 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 Prime 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 10-14: Synchronous Reset

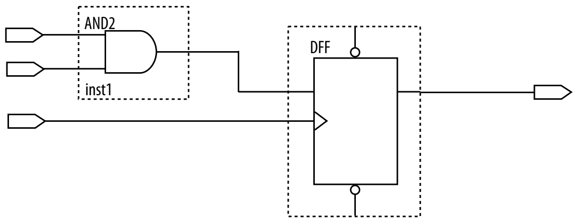
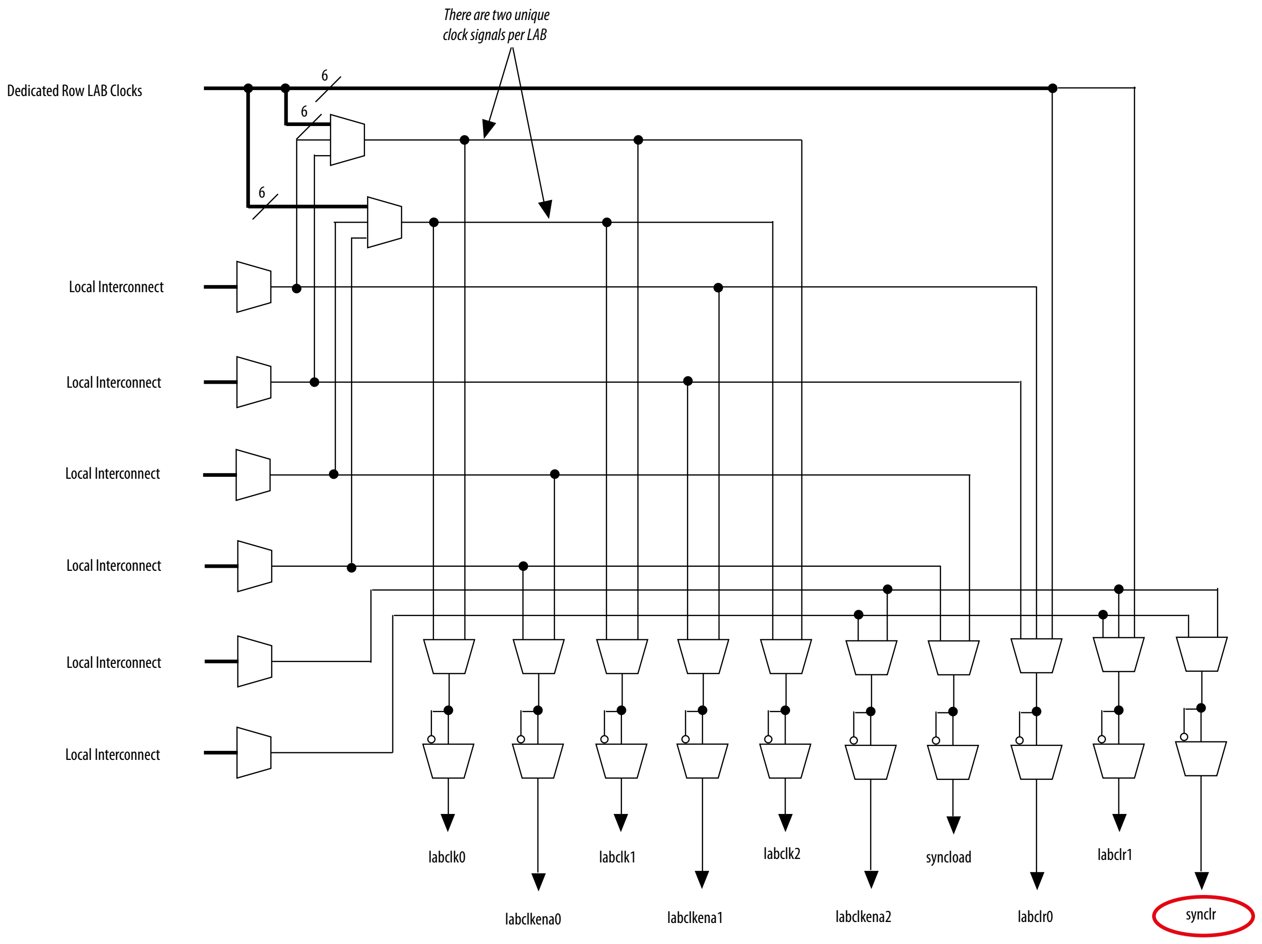
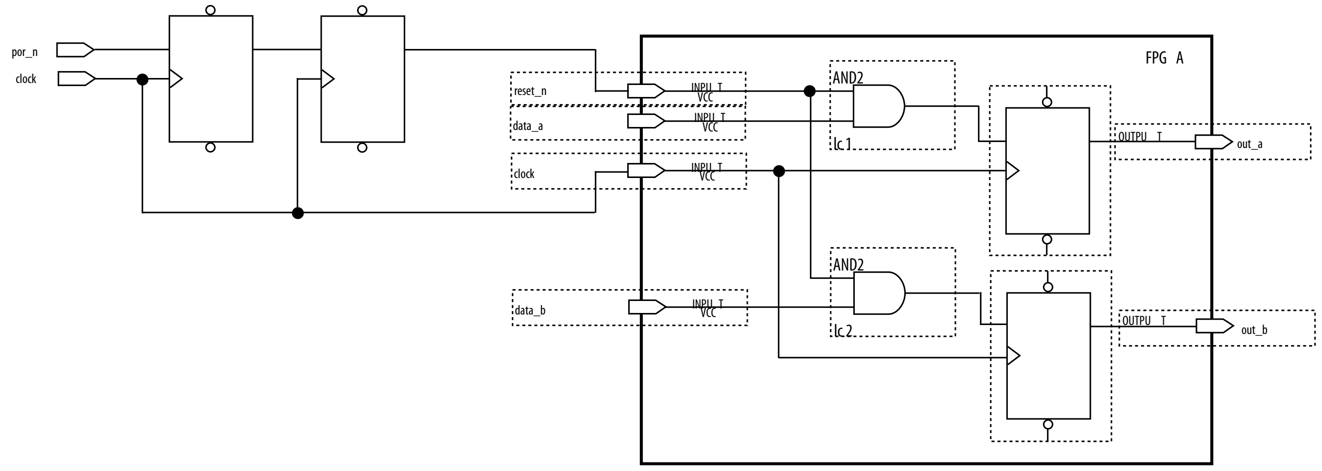


Figure 10-15: LAB-Wide Control Signals



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 10-16: 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 10-6: 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
```

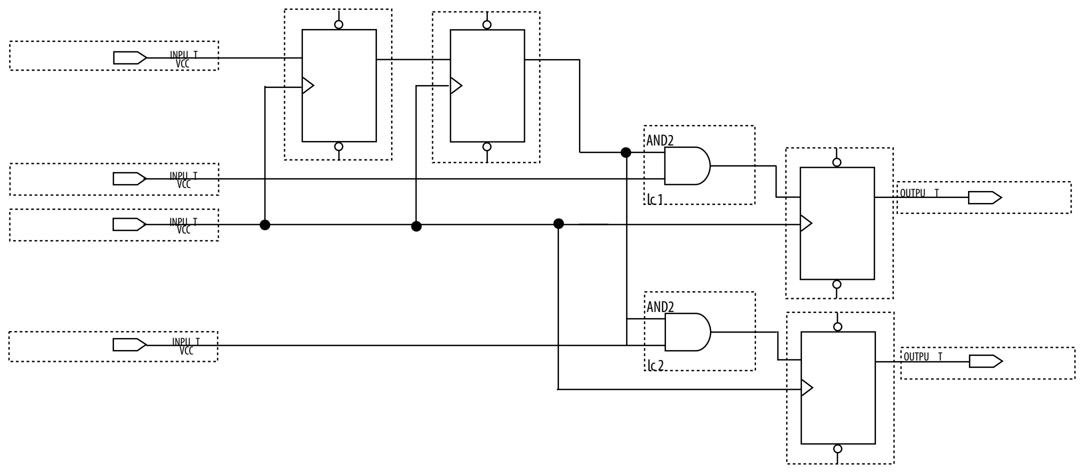
The following example shows the constraints for the externally synchronous reset. Because the external reset is synchronous, you only need to constrain the `reset_n` signal as a normal input signal with `set_input_delay` constraint for `-max` and `-min`.

Example 10-7: 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 10-17: 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 10-8: 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
```

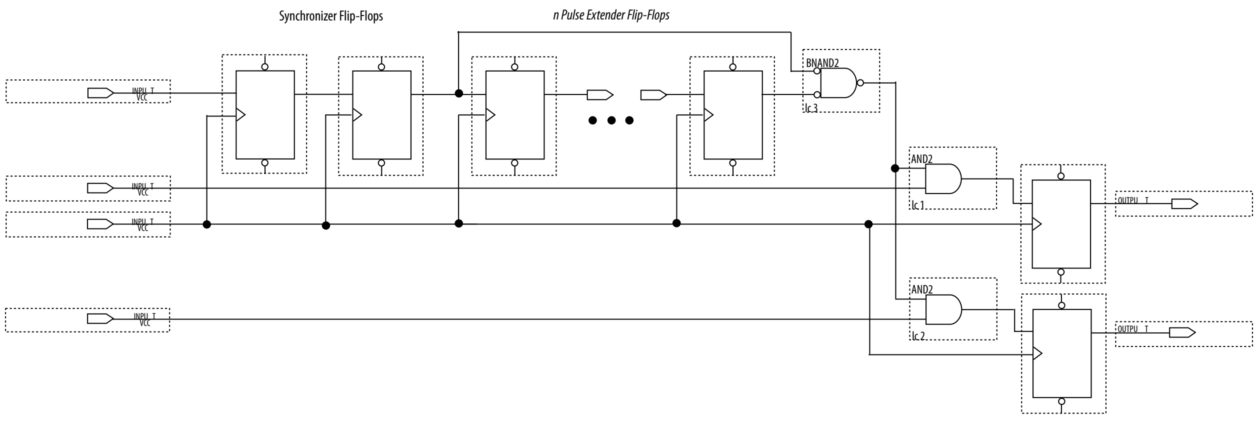
The SDC constraints are similar to the external synchronous reset, except that the input reset cannot be constrained because it is asynchronous and should be cut with a `set_false_path` statement to avoid these being considered as unconstrained paths.

Example 10-9: 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 10-18: 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.

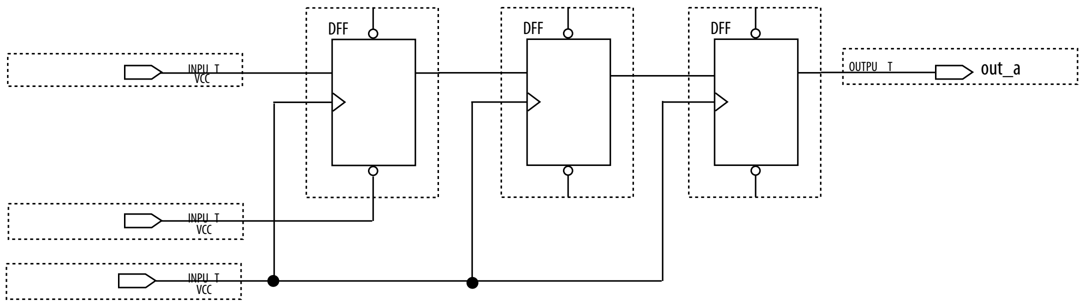
Using Asynchronous Resets

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 10-19: 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 10-10: 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 10-11: 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.

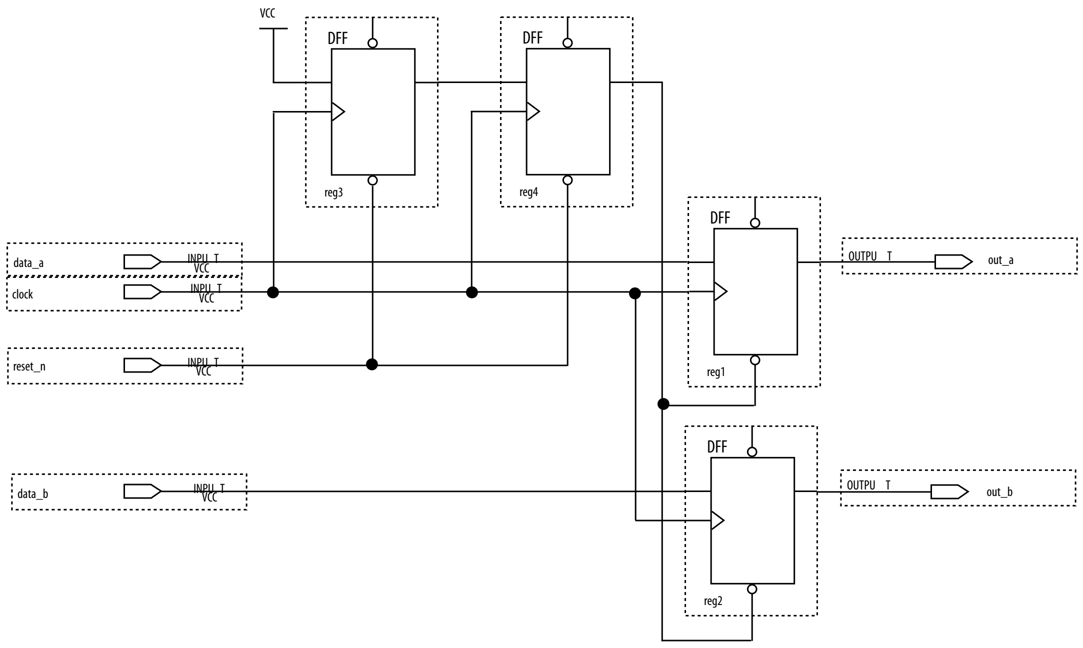
Use Synchronized Asynchronous Reset

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 10-20: 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 10-12: 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 Prime 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 10-1: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
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 .

Date	Version	Changes
December 2010	10.1.0	• Title changed from Design Recommendations for Altera Devices and the Quartus Prime Design Assistant. • Updated to new template. • Added references to Quartus Prime Help for "Metastability" on page 9–13 and "Incremental Compilation" on page 9–13. • Removed duplicated content and added references to Quartus Prime Help for "Custom Rules" on page 9–15.
July 2010	10.0.0	• Removed duplicated content and added references to Quartus Prime 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 11

2015.11.02

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 Prime integrated synthesis and other EDA tools), refer to the tool vendor's documentation.

Related Information

- Recommended Design Practices on page 10-1
- Advanced Synthesis Cookbook
- Design Examples
- Reference Designs
- Quartus Prime Integrated Synthesis

Using Provided HDL Templates

You can use provided HDL templates to start your HDL designs.

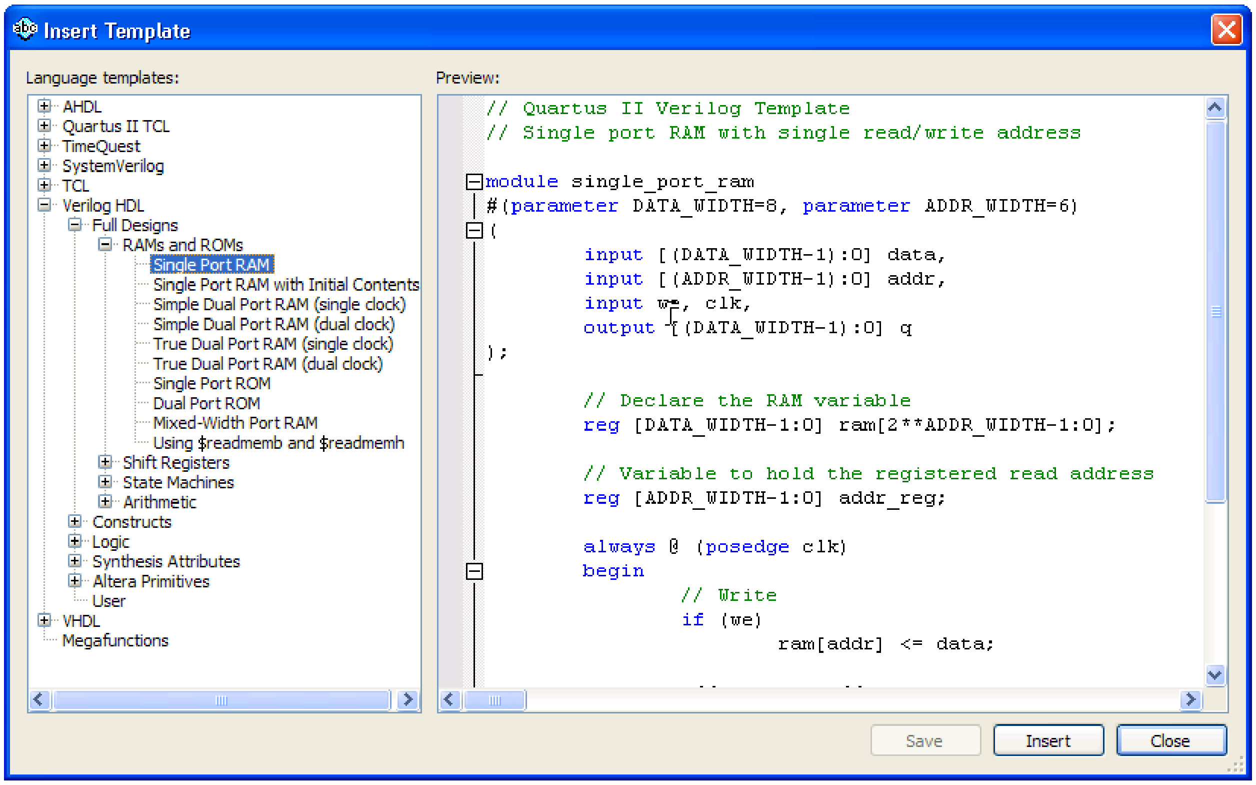
Altera provides templates for Verilog HDL, SystemVerilog, and VHDL. Many of the HDL examples in this document correspond with the**Full Designs** examples in the **Quartus Prime Templates**. You can insert HDL code into your own design using the templates or examples.

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 11-1: Inserting a RAM Template



```
// 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 Prime 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 Prime 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 Prime 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 11-3
- **Inferring Memory Functions from HDL Code** on page 11-8
- **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 Prime 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 11-1: 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 11-2: 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 11-3: 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 11-4: 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 Prime 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 Prime 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 11-5: 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 11-6: 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 11-7: 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 11-8: 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 Prime templates provided in the Quartus Prime software as a starting point.

Most of these designs can also be found on the Design Examples page on the Altera website.

Table 11-1: 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 Prime 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 Prime 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 Prime 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 Prime 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 11-9: 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 11-10: 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 11-25

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 Prime 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 Prime 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 Prime Integrated Synthesis

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 Prime 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 Prime 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 Prime 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 Prime Integrated Synthesis

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 11-11: 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 11-12: 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 11-12
- **Single-Clock Synchronous RAM with New Data Read-During-Write Behavior** on page 11-15

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 Prime 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 11-13: 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 11-14: 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 11-15: 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 Prime 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 11-12
- **Check Read-During-Write Behavior** on page 11-12
- **Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior** on page 11-14

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 Prime integrated synthesis infers this type of RAM, it issues a warning because of the undefined read-during-write behavior.

Example 11-16: 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 11-17: 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 11-12

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 Prime 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 Prime 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 11-18: 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 11-19: 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 11-20: 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 11-21: 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 11-12

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 Prime 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 Prime 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 11-22: 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 11-23: 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 11-24: 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 11-25: 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 Prime integrated synthesis.

Refer to the Quartus Prime 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 11-26: 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 11-27: 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 Prime 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 Prime integrated synthesis automatically converts the initial block into a **.mif** file for the inferred RAM.

Example 11-28: 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 Prime 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 11-29: 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 Prime integrated synthesis automatically converts the default value into a .mif file for the inferred RAM.

Example 11-30: 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 Prime Integrated Synthesis

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 Prime 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 Prime 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 Prime 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 Prime Help explains the condition under which the functionality changes when you use Quartus Prime integrated synthesis.

The following ROM examples map directly to the Altera memory architecture.

Example 11-31: 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 11-32: 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 11-33: 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 11-34: 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 Prime Integrated Synthesis

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 Prime 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 Prime 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 Prime integrated synthesis uses the following guidelines which are common in other EDA tools. The Quartus Prime 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 Prime 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 11-35: 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 11-36: 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 11-37: 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 11-38: 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 Prime Design Assistant** on page 10-1

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 Prime 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 Prime 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 Prime integrated synthesis converts default values for registered signals into **Power-Up Level** settings. When the Quartus Prime software reads the default values, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

Example 11-39: 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 11-40: 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 Prime 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 Prime Integrated Synthesis**

Secondary Register Control Signals Such as Clear and Clock Enable

The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells.

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 11-41: 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 11-42: 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 11-43: 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 10-1

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 11-44: 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 Prime Integrated Synthesis

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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 11-45: 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 11-46: 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 Prime software:

Example 11-47: 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 Prime 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 Prime software. If a third-party synthesis tool implements a latch using the Altera latch IP core, the Quartus Prime 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 Prime 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 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

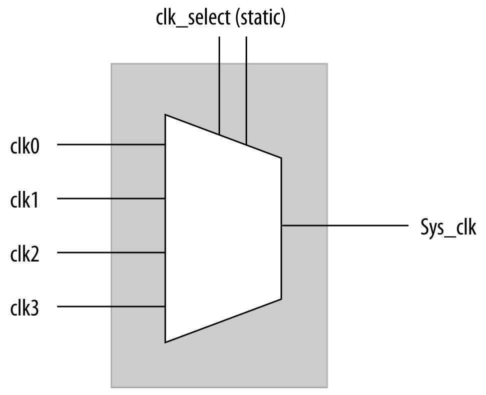
Clock multiplexing is sometimes used to operate the same logic function with different clock sources.

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.

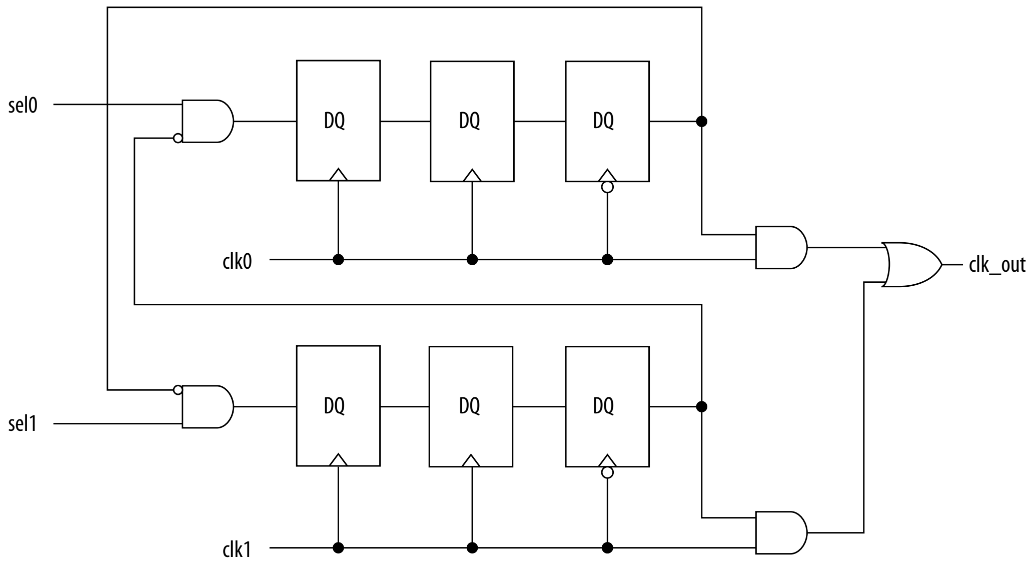
Figure 11-2: Simple Clock Multiplexer in a 6-Input LUT



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 11-3: 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 11-48: 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 Prime 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 11-49: 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 Prime 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 Prime 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 Prime Integrated Synthesis

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 Prime 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 Prime 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 Prime 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 Prime software.
- No state machine is inferred in the Quartus Prime 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 Prime software if the state variable is an output.
- No state machine is inferred in the Quartus Prime software for signed variables.

Related Information

- Verilog-2001 State Machine Coding Example on page 11-47
- SystemVerilog State Machine Coding Example on page 11-49

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 11-50: 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 Prime 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 Prime integrated synthesis synthesizes the design, but does not infer or optimize the logic as a state machine.

Example 11-51: 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 Prime 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 Prime 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 11-52: 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 Prime Software Option for Multiplexer Restructuring

Quartus Prime 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 Prime-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 Prime Integrated Synthesis

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 11-53: 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 Prime Synthesis will try to combine the selector and decoder into a binary multiplexer.

Example 11-54: 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.

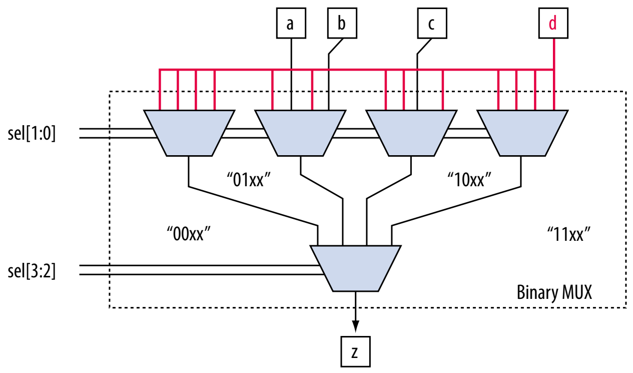
These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL.

Example 11-55: 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 11-4: 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 Prime 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:

- Synthesize each CRC block as a separate project in your third-party synthesis tool and then write a separate Verilog Quartus Mapping (**.vqm**) or EDIF netlist file for each.

Take Advantage of Latency if Available

If your design can use more than one cycle to implement the CRC functionality, adding registers and retiming the design can help reduce area, improve performance, and reduce power utilization.

If your synthesis tool offers a retiming feature (such as the Quartus Prime 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 11-35

- **Designing with Low-Level Primitives User Guide**

Comparator HDL Guidelines

Synthesis software, including Quartus Prime 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 Prime 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):

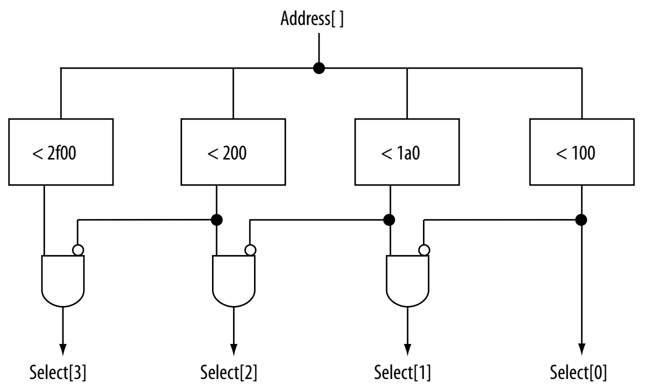
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

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 11-5: 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 Prime 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 Prime 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 11-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
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 Prime 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 Handbook Archive

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

Design Compilation 12

2015.11.02

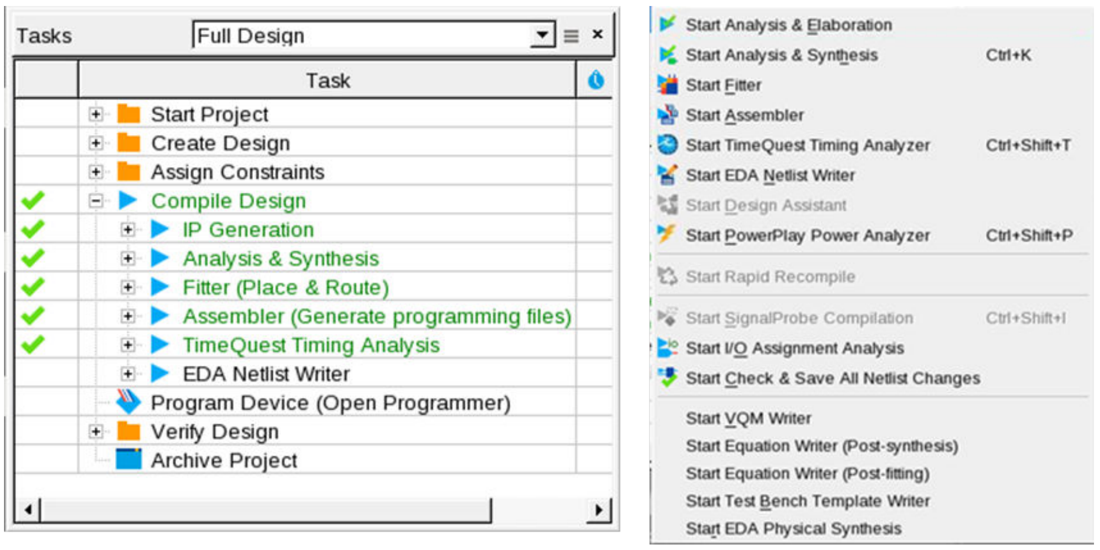
QPP5V1 Subscribe Send Feedback

The Quartus Prime Compiler is a set of independent modules that analyze, synthesize, place, and route your project design files and ultimately generate programming files targeting an Altera device. The Compiler supports a wide variety of high-level, RTL, and schematic design entry methods.

Prior to running the Compiler, click **Assignments** > **Settings** to specify options that effect the design compilation. Once you complete a stage of your design, run one or more Compiler modules to implement your design changes and review resulting reports. Click **Processing** > **Start Compilation** to run a full compilation, or run any of the individual Compiler modules from the same menu:

- IP Generation—checks the status of IP variations in your project
- Analysis & Synthesis—verifies syntax, builds design database(s), synthesizes logic, and technology mapping
- Fitter—determines specific logic placement and routing in the target device
- Assembler—generates a device programming image
- TimeQuest Timing Analyzer— validates design timing performance
- EDA Netlist Writer—generates output netlist files for use with other EDA tools

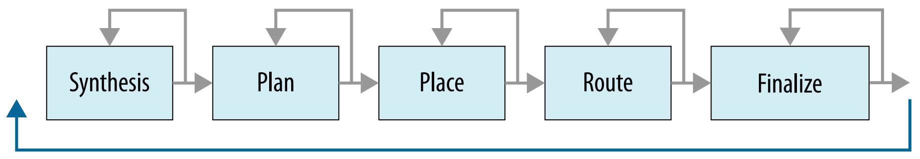
Figure 12-1: Full Design Compilation Flows and Commands


Tasks
Full Design
Task
Start Project
Create Design
Assign Constraints
Compile Design
IP Generation
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
TimeQuest Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)
Verify Design
Archive Project
Start Analysis & Elaboration
Start Analysis & Synthesis Ctrl+K
Start Fitter
Start Assembler
Start TimeQuest Timing Analyzer Ctrl+Shift+T
Start EDA Netlist Writer
Start Design Assistant
Start PowerPlay Power Analyzer Ctrl+Shift+P
Start Rapid Recompile
Start SignalProbe Compilation Ctrl+Shift+I
Start I/O Assignment Analysis
Start Check & Save All Netlist Changes
Start VQM Writer
Start Equation Writer (Post-synthesis)
Start Equation Writer (Post-fitting)
Start Test Bench Template Writer
Start EDA Physical Synthesis

The Quartus Prime Pro Edition introduces compilation with the Spectra-Q engine. The Spectra-Q engine is an infrastructure upgrade that supports next generation synthesis, physical optimizations, design methodologies, and device architectures. The Spectra-Q engine helps to streamline the FPGA development process, ensures the highest performance for the least effort, and supports the latest Altera devices with the following emerging features:

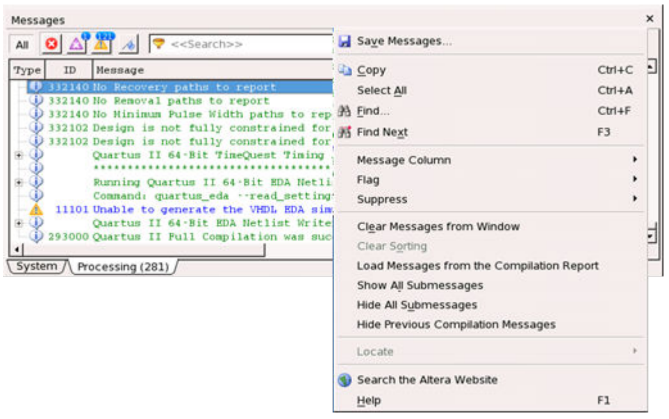
- Hierarchical Project Database—The Spectra-Q engine creates a separate compilation database for each design partition. That hierarchical database preserves individual post-synthesis, post-placement, and post-place & route results for each partition. Isolating each partition allows optimization without impacting other partition placement or routing.
- Spectra-Q Synthesis—integrates new, stricter language parser supporting all major IEEE RTL languages, with enhanced algorithms, and parallel synthesis capabilities.
- Spectra-Q Physical Synthesis Optimization—performs combinational and sequential optimization during fitting to improve circuit performance.
- Rapid Recompile—automatically reuses previous verified results to reduce compile time and while preserving timing.
- Fitter Stage Reporting—reports now include detailed information for each *stage* of the place and route process

Figure 12-2: Spectra-Q Compilation Stages



During compilation processing, the Messages window dynamically updates to display Information, Warning, and Error messages about your project. Review any warnings and correct any errors to complete successful compilation. The Compilation report window displays detailed, actionable reports about your design processing and results.

Figure 12-3: Messages Window



```
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
```

Related Information

- **Migrating to Quartus Prime Pro Edition** on page 1-3

- **Compiling Designs** on page 12-13
- **Reducing Compilation Process Time** on page 12-31

Spectra-Q Compiler

The Spectra-Q Compiler is a suite of separate executable modules that function in concert to optimize and convert your project design files to Altera device programming files. Each Compiler module performs specific optional or required functions in the compilation process.

You can run a full compilation, or you can run the individual Compiler modules separately. The following Compiler modules run during a full compilation:

Table 12-1: Compiler Stages

Compiler Module	Function
Analysis & Elaboration	Checks for design file and project errors and builds a one or more design databases. Elaborates all files in the top-level entity hierarchy.
Analysis & Synthesis	After running Analysis & Elaboration, Analysis & Synthesis performs logic synthesis to optimize, minimize, and technology map design logic to device resources. Analysis & Synthesis provides comprehensive and standards-compliant Verilog HDL, VHDL, and SystemVerilog language support. Quartus Prime Pro Edition software introduces synthesis with the Spectra-Q engine (Spectra-Q Synthesis).
Fitter (Place & Route)	After running Analysis & Synthesis, the Fitter assigns the placement and routing of the design to specific device resources, while honoring timing and placement constraints. Quartus Prime Pro Edition software introduces a hybrid placement technique that combines analytical and annealing placement techniques. The Fitter also reports detailed data for each stage of place and route.
Assembler	After running the Fitter, the Assembler converts the Fitter's placement and routing assignments into a programming image for the device.
TimeQuest Timing Analyzer	After running Analysis & Synthesis or the Fitter, TimeQuest analyzes, debugs, and validates the timing performance of all logic in a design.
EDA Netlist Writer	After running Analysis & Synthesis or the Fitter, EDA Netlist Writer optionally generates output netlist files for use with other EDA tools.

Running a full compilation on a large design can be time consuming. As your design develops, you can run Compiler modules separately to process and optimize design elements. When your design is complete and you are ready to program a device, run a full compilation to include all Compiler, timing analysis, and programming file generation modules.

Related Information

- **Hierarchical Project Database** on page 12-4
- **Spectra-Q Design Synthesis** on page 12-4
- **Design Place and Route** on page 12-11
- **Compiling Designs** on page 12-13

- **Migrating to Quartus Prime Pro Edition** on page 1-3

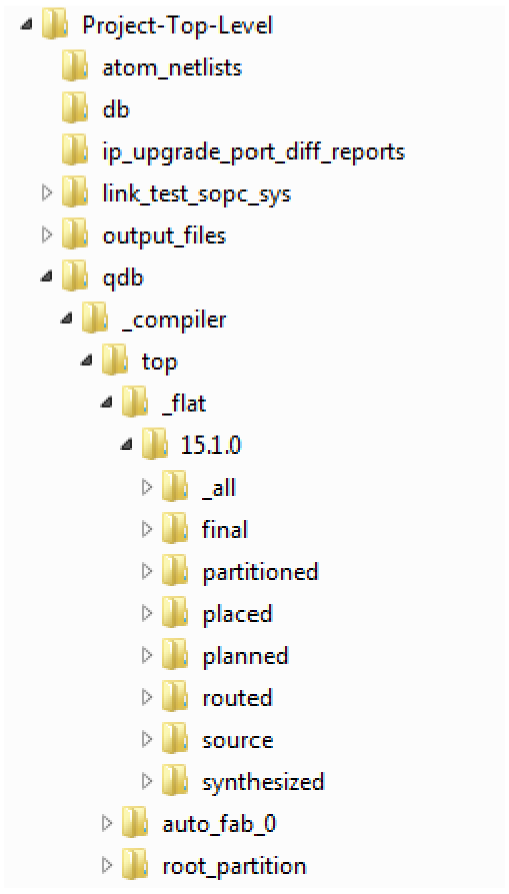
Hierarchical Project Database

Spectra-Q compilation generates a new hierarchical database infrastructure that isolates the compilation results of each design partition.

This hierarchical structure allows you to optimize specific design elements without impacting placement and routing in other partitions. The Spectra-Q Compiler fully preserves routing and placement within a partition. Changes to other portions of the design hierarchy impact routing only outside the partition.

The isolation and portability of hierarchical project database also supports distributed work groups and compilation processing across multiple machines.

Figure 12-4: Project Database Structure



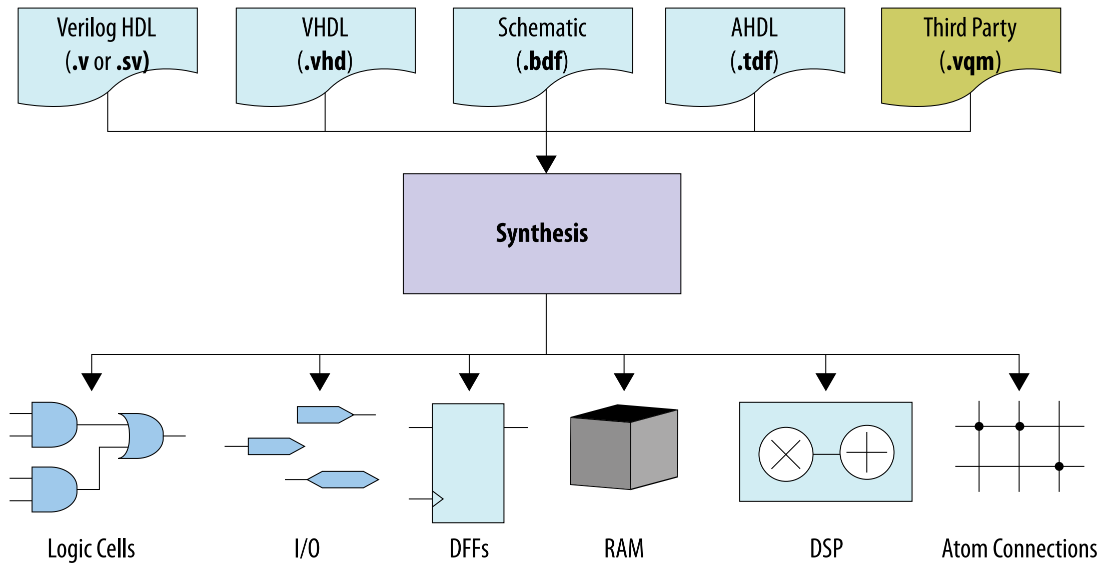
```
Project-Top-Level
    atom_netlists
    db
    ip_upgrade_port_diff_reports
    link_test_sopc_sys
    output_files
    qdb
        _compiler
            top
                _flat
                    15.1.0
                        _all
                        final
                        partitioned
                        placed
                        planned
                        routed
                        source
                        synthesized
                auto_fab_0
                root_partition
```

Spectra-Q Design Synthesis

Design synthesis is the process of translating design source files into an atom netlist. The Quartus Prime software synthesizes standard Verilog HDL, VHDL, and SystemVerilog design files. In addition, synthesis processes Quartus Prime schematic editor files, and accepts Verilog Quartus Mapping (**.vqm**) 3rd party tool files.

The Analysis & Synthesis module of the Compiler analyzes and synthesizes design files and creates one or more project databases for each design partition. You can use different synthesis flows and specify various settings to fine-tune your synthesis processing.

Figure 12-5: Design Synthesis



The Quartus Prime Pro Edition software includes design synthesis with the Spectra-Q engine. Spectra-Q synthesis includes a new front-end language parser that supports synthesis of standards-compliant VHDL, Verilog HDL, and SystemVerilog design files. Spectra-Q language syntax checking is more strict that other Quartus software products. Spectra-Q synthesis (`quartus_syn`) replaces Quartus Integrated Synthesis (`quartus_map`) found in other Quartus software products.[8]

- Faster logic synthesis algorithms and true parallel synthesis
- Improved language support for all IEEE RTL languages, including expansive support for SystemVerilog-2005 and VHDL-2008
- New RAM inference engine infers RAMs from GENERATE statements or array of integers
- Stricter syntax and semantics check for standards compliance and improved compatibility with other EDA tools

Synthesis Methodology

Synthesis examines the logical completeness and consistency of the design, and checks for boundary connectivity and syntax errors. Synthesis minimizes and optimizes logic of your design.

For example, synthesis infers flipflops, latches and state machines from "behavioral" languages, such as Verilog HDL, VHDL, and SystemVerilog. Synthesis may replace operators, such as + or -, with modules from the Altera IP Library, when advantageous. During synthesis, the Compiler may change or remove

[8] For brevity, this document refers to Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software collectively as "other Quartus software products."

user logic and or design nodes. Spectra-Q synthesis minimizes gate count, removes redundant logic, and ensures efficient use of device resources.

Successful design synthesis produces an atom netlist. Atom refers to the most basic hardware resource in the FPGA device. Atoms include logic cells organized into lookup tables, D flip-flops, I/O pins, block memory resources, DSP blocks, and the connections required to connect atoms. The atom netlist is a database of the atom elements that synthesis selects to implement the design.

You can start a full compilation, which includes the Analysis & Synthesis module, or you can start click **Processing** > **Start** > **Start Analysis & Synthesis** to run synthesis independently. The Compiler generates detailed reports following synthesis in the **Synthesis** report.

You can use Analysis & Synthesis to perform the following compilation processes:

Table 12-2: Analysis and Synthesis Processes

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, click **Processing** > **Analyze Current File**.
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 **Processing** > **Start** > **Start Analysis & Elaboration**.
Hierarchy Elaboration	Parses HDL designs and generates a skeleton of hierarchies. Hierarchy Elaboration is similar to the Analysis & Elaboration, 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, click **Processing** > **Start** > **Start Analysis & Synthesis**.

Verilog and SystemVerilog Synthesis Support

The Analysis & Synthesis Compiler 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 Quartus Prime 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 Prime software support for Verilog HDL is case sensitive in accordance with the Verilog HDL standard. The Quartus Prime software supports the compiler directive `` `define``, in accordance with the Verilog HDL standard.

The Quartus Prime 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 Prime software initially searches relative to the project directory. If the Quartus Prime 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. Spectra-Q synthesis searches for all modules or entities earlier in the synthesis process than other Quartus software tools, which may result in more early syntax errors for undefined entities.

Related Information

- **Verilog HDL Input Settings (Settings Dialog Box)** on page 12-18
- **Migrating to Quartus Prime Pro Edition** on page 1-3
- **Recommended Design Practices** on page 10-1
- **Recommended HDL Coding Styles** on page 11-1

Design Libraries

By default, the Quartus Prime software compiles all design files into one or more 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:

When compiling a design instance, the Quartus Prime 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 Prime software could not locate the entity definition, the software searches for a unique entity definition in all design libraries. If the Quartus Prime 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.

Note: Spectra-Q synthesis searches for all modules or entities earlier in the synthesis process than other Quartus software tools, which may result in more early syntax errors for undefined entities.[9]

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**.

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.

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 1**.

[9] For brevity, this section refers to Quartus Prime Standard Edition, Quartus Prime Lite Edition, and Quartus II software collectively as "other Quartus software products."

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.

Initial Constructs and Memory System Tasks

The Quartus Prime software infers power-up conditions from the Verilog HDL `initial` constructs. The Quartus Prime software also creates power-up settings for variables, including RAM blocks. If the Quartus Prime 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 Prime 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 12-1: 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.

Example 12-2: 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
- Power-Up Level

Verilog HDL Macros

The Quartus Prime 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 Prime software or on the command line.

VHDL Synthesis Support

The Analysis & Synthesis Compiler 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 Prime 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.

Related Information

- VHDL Input Settings (Settings Dialog Box) on page 12-17
- Migrating to Quartus Prime Pro Edition on page 1-3

VHDL Standard Libraries and Packages

The Quartus Prime 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 Prime 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` in the ALTERA library
- Altera IP core packages `altera_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.

VHDL wait Constructs

The Quartus Prime software supports one VHDL `wait until` statement per process block. However, the Quartus Prime 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;
```

Timing-Driven Synthesis

Timing-driven synthesis accounts for any **.sdc** timing constraints in optimizing the circuit during synthesis. Timing analysis runs first to obtain timing information about the netlist. Synthesis then focuses on performance for timing-critical design elements, while optimizing non-timing-critical portions for area. Synthesis preserves SDC constraints and does not perform timing optimizations that conflict with **.sdc** timing constraints. Duplicate registers must have compatible timing constraints.

Spectra-Q synthesis includes timing-driven synthesis by default. You can enable or disable this option by clicking **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.

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** option. Runtime and peak memory use increases slightly if you turn on the **Timing-Driven Synthesis** option.

The **Timing-Driven Synthesis** logic option impacts the **Optimization Technique** option:

- **Optimization Technique Speed**—optimizes timing-critical portions of your design for performance at the cost of increasing area (logic and register utilization)
- **Optimization Technique Balanced**—also optimizes the timing-critical portions of your design for performance, but the option allows only limited area increase
- **Optimization Technique Area**—optimizes your design only for area

Even with the **Optimization Technique** 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.

To turn on or turn off the **Timing-Driven Synthesis** option, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)**.
2. Turn on or turn off **Timing-Driven Synthesis**.

Note: 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.

Design Place and Route

The Fitter module (`quartus_fit`) of the Compiler performs design place and route. During place and route, the Fitter determines the best placement and routing of logic in the target Altera device, with respect to your settings and constraints.

The Fitter accounts for timing requirements and other constraints while assigning each logic function to device resources. The Fitter selects appropriate interconnection paths and pin assignments. If you assign logic to specific device resources, the Fitter attempts to match those requirements, and then fits and optimizes any remaining unconstrained design logic. If the Fitter cannot fit the design in the current target device, the Fitter terminates compilation and issues an error message.

The Spectra-Q Compiler introduces a hybrid placement technique that combines analytical and annealing placement techniques. Analytical placement determines an initial mathematical starting placement. The annealing technique then fine tunes logic block placement in high resource utilization scenarios.

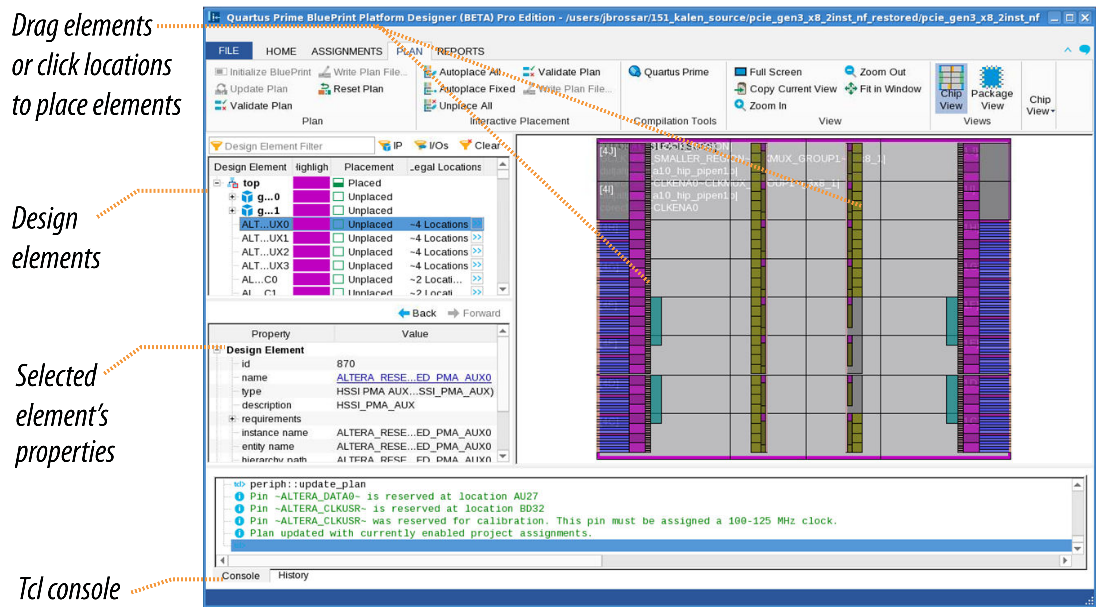
You can start a full compilation, which includes the Fitter module, or click **Processing** > **Start** > **Start Fitter** to run the Fitter independently. You must run synthesis before running the Fitter. The Compiler generates detailed reports following place and route in the **Fitter** reports. The Fitter reports details for the **Plan**, **Place**, **Route**, and **Finalize** stages of place and route.

Optimizing Place and Route

You can specify various settings and constraints to influence place and route. To achieve an optimal fit and timing closure for your design, assign logic to appropriate physical device resources, such as I/O pins, logic cells, or LABs using any of the following methods:

- **Assignments** > **Assignment Editor**—spreadsheet-like interface for assigning logic and other options to device resources such as nodes and buses.
- **Assignments** > **Pin Planner**—helps you visualize, plan, and assign device I/O pins in a graphical view of the target device package.
- **Tools** > **BluePrint Platform Designer**—helps you to accurately plan constraints for design implementation, such as constraining I/O pins, PLLs, and clocks, as well as I/O and HSSI external interfaces. Use BluePrint to prototype interface implementations and rapidly define a legal device floorplan.
- **Tools** > **Chip Planner**—visual display of logic placement, LogicLock Plus regions, relative resource usage, detailed routing information, fan-ins and fan-outs, paths between registers, and high-speed transceiver channels. You can view physical timing estimates, routing congestion, and clock regions.

Figure 12-6: BluePrint Platform Designer GUI



Design Element	Highlight	Placement	Legal Locations
top		Placed	
g...0		Unplaced	
g...1		Unplaced	
ALT...UX0		Unplaced	~4 Locations
ALT...UX1		Unplaced	~4 Locations
ALT...UX2		Unplaced	~4 Locations
ALT...UX3		Unplaced	~4 Locations
AL...C0		Unplaced	~2 Locati...

Property	Value
Design Element	
id	870
name	ALTERA_RESE...ED_PMA_AUX0
type	HSSI PMA AUX...SSI_PMA_AUX)
description	HSSI_PMA_AUX
requirements	
instance name	ALTERA_RESE...ED_PMA_AUX0
entity name	ALTERA_RESE...ED_PMA_AUX0

```
tcl> periph::update_plan
Pin ~ALTERA_DATA0~ is reserved at location AU27
Pin ~ALTERA_CLKUSR~ is reserved at location BD32
Pin ~ALTERA_CLKUSR~ was reserved for calibration. This pin must be assigned a 100-125 MHz clock.
Plan updated with currently enabled project assignments.
```

To set Fitter settings, click **Assignments** > **Settings** > **Compiler Settings**. You can select an **Optimization Mode** to direct the Compiler to emphasize specific design metrics, including increased performance at the cost of increased compilation time. The **Advanced Fitter Settings** allow you to fine tune Fitter processing to meet your design goals. These settings also have some influence over synthesis processing.

The Fitter includes advanced optimization algorithms to help you achieve timing closure, optimize area, and reduce power consumption. The various Fitter settings control the behavior of these algorithms. Because each design has unique characteristics, each design may benefit from different optimal settings. Click **Tools** > **Design Space Explorer II** to explore the range of settings that achieves the best possible fit for your design. DSE II determines the best collection of setting based on your optimization goals. You can specify whether DSE II optimizes for speed, area, or power.

Related Information

- BluePrint Design Planning
- Constraining Designs
- Managing Device I/O Pins
- Managing Quartus Prime Projects on page 2-1
- Analyzing the Design Floorplan with the Chip Planner

Spectra-Q Physical Synthesis Optimization

The Quartus Prime software includes advanced physical synthesis optimization. The **Spectra-Q Physical Synthesis** option uses the Spectra-Q engine to perform combinational and sequential optimization during fitting to improve circuit performance for Arria 10 designs.

Spectra-Q Physical Synthesis fine tunes the physical placement and structure of logic to achieve timing closure and meet performance requirements. Enable the Spectra-Q physical synthesis option by clicking **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**.

Programming File Generation

The Assembler compilation module generates a device programming image and other optional programming files when your design is complete. For Altera FPGAs, this programming image is in the form of one or more Programmer Object Files (**.pof**), SRAM Object Files (**.sof**), Hexadecimal (Intel-Format) Output Files (**.hexout**), Tabular Text Files (**.ttf**), and Raw Binary Files (**.rbf**).

The Assembler also generates detailed data for the PowerPlay Power Analyzer. You can customize the Assembler by clicking **Assignments** > **Device** > **Device and Pin Options**. For example, you can make the following settings in the **Device and Pin Options** dialog box:

- Auto usercode or JTAG user code
- Configuration scheme
- Optional programming files
- Reserve unused pins options
- Adjust the voltage of the LVTTL/LVCMOS pins
- Override electromigration default values

The Assembler report files display any messages the Assembler generates. The Status window and the Flow Elapsed section of the Report window record the time spent processing in the Assembler during project compilation.

Compiling Designs

The Quartus Prime Compiler synthesizes the logic of your design and generates a programming file for implementation in an Altera device. Alternatively, you can run Compiler modules individually from the Processing menu.

The following process describes the high level steps in a typical full compilation:

1. Create or open a Quartus Prime project with valid design files for compilation.
2. Before running modules of the Compiler, specify basic settings that effect the compilation:

   - **Assignments** > **Device**—specify the target Altera device and options.
   - **Assignments** > **Device** > **Device and Pin Options**—specify device and pin options for the target Altera device
   - **Assignments** > **Settings** > **Compilation Process Settings**—specify options that effect compilation processing time, parallel compilation, recompilation, and netlist preservation
   - **Assignments** > **Settings** > **Compiler Settings**—specify synthesis optimization goals and other **Advanced Settings** for synthesis and fitting.

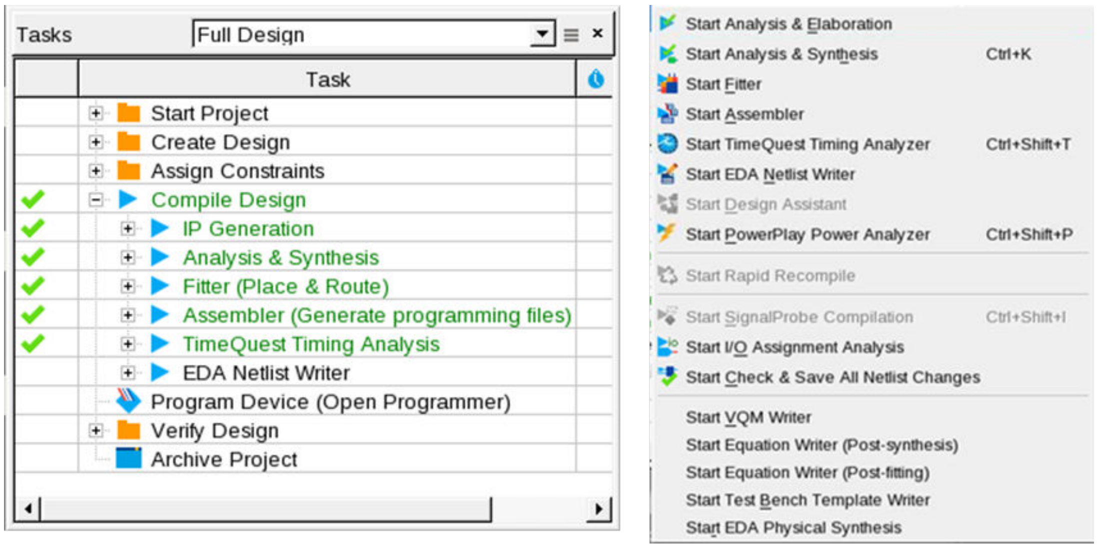
- **Assignments** > **Settings** > **Assembler**—specify options for programming file generation.
- **File** > **Convert Programming Files**—specify options to convert programming files for use by systems such as embedded processors.
- **Tools** > **TimeQuest Timing Analyzer**—specify required timing conditions for proper operation of your design.

3. Plan and specify placement of physical device resources:

- **Tools** > **BluePrint Platform Designer**—plan interfaces and device periphery.
- **Assignments** > **Pin Planner**—edit, validate, or export pin assignments.

4. Click **Processing** > **Start Compilation** to run all Compiler modules defined in the current Compiler flow. Alternatively, run any Compiler module individually from the same menu.

Figure 12-7: Full Design Compilation Flow


Tasks
Full Design
Task
Start Project
Create Design
Assign Constraints
Compile Design
IP Generation
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
TimeQuest Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)
Verify Design
Archive Project
Start Analysis & Elaboration
Start Analysis & Synthesis Ctrl+K
Start Fitter
Start Assembler
Start TimeQuest Timing Analyzer Ctrl+Shift+T
Start EDA Netlist Writer
Start Design Assistant
Start PowerPlay Power Analyzer Ctrl+Shift+P
Start Rapid Recompile
Start SignalProbe Compilation Ctrl+Shift+I
Start I/O Assignment Analysis
Start Check & Save All Netlist Changes
Start VQM Writer
Start Equation Writer (Post-synthesis)
Start Equation Writer (Post-fitting)
Start Test Bench Template Writer
Start EDA Physical Synthesis

Related Information

- BluePrint Design Planning
- The TimeQuest Timing Analyzer
- Managing Device I/O Pins
- Advanced Synthesis Settings Reference on page 12-34
- Advanced Fitter Settings Reference on page 12-42

Compiler Flows

Quartus Prime software provides predefined compilation flows that you can start with a single command. You can also define and save a custom compilation flow to match your design scenario. You can select, launch, or define compilation flow in the Tasks window or at the command line with the `quartus_sh --flow` executable.

When you select a predefined compilation flow, the Tasks window displays a list of prompts that you can click to execute actions in the flow. When you click **Processing** > **Start Compilation**, the Compiler starts the currently selected flow automatically.

Table 12-3: Compiler Flows (Part 1 of 2)

Available Flows	Flow Sequence
Full Design	1. Start Project Prompts 2. Create Design Prompts 3. Runs IP Generation 4. Runs Analysis & Synthesis 5. Runs Fitter 6. Runs Assembler 7. Runs TimeQuest 8. Runs EDA Netlist Writer 9. Design Simulation Prompts 10. On-Chip Debugging Prompts 11. Run Power Analyzer 12. Run SSN Analyzer 13. Engineering Change Order Prompts
Compilation	1. Runs IP Generation 2. Runs Analysis & Synthesis 3. Runs Fitter 4. Runs Assembler 5. Runs TimeQuest 6. Runs EDA Netlist Writer

Table 12-4: Compiler Flows (Part 2 of 2)

Available Flows	Flow Sequence
Gate-level Simulation	1. Runs Analysis & Synthesis 2. Runs Fitter 3. Runs TimeQuest 4. Runs EDA Netlist Writer 5. Launches your Gate-level simulation tool
RTL Simulation	1. Runs Analysis & Elaboration 2. Launches your Gate-level simulation tool
Rapid Recompile	1. Runs Rapid Recompile Analysis & Synthesis 2. Runs Rapid Recompile Fitter 3. Runs Assembler 4. Runs TimeQuest

Running Spectra-Q Synthesis

The Compiler's Analysis & Synthesis module performs logic synthesis, minimization, and technology mapping to device resources. Quartus Prime Pro Edition automatically uses Spectra-Q engine synthesis (`quartus_syn`) to synthesis the RTL in your design files. Prior to running Analysis and Synthesis, you can plan and assign placement of important physical device resources, and specify a broad range of settings that customize design synthesis:

- Specify the design files in your project
- Specify the HDL language version for synthesis
- Select options for setting attributes without modifying source code
- Specify general synthesis performance, power, or logic usage goals
- Assign project-wide default parameters
- Fine tune synthesis with advanced options

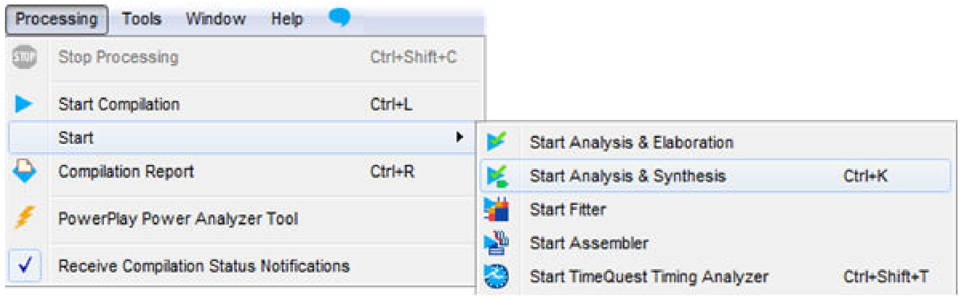
When you are ready to run synthesis, click **Processing** > **Start** > **Start Analysis & Synthesis**. Alternatively, you can start synthesis in batch mode at the Unix command line by running the following command.

```
quartus_syn <project name> [<options>]
```

To list all command options, type: `quartus_syn -h`.

The Compiler confirms that prerequisite modules have run, and launches the Spectra-Q synthesis module to synthesis design logic. The Messages window dynamically displays processing information, warnings, or errors. Following Analysis and Synthesis processing, the Synthesis report provides detailed information about synthesis of each design partition.

Figure 12-8: Start Spectra-Q Synthesis (quartus_syn)


Processing
Tools
Window
Help
Stop Processing Ctrl+Shift+C
Start Compilation Ctrl+L
Start
Compilation Report Ctrl+R
PowerPlay Power Analyzer Tool
Receive Compilation Status Notifications
Start Analysis & Elaboration
Start Analysis & Synthesis Ctrl+K
Start Fitter
Start Assembler
Start TimeQuest Timing Analyzer Ctrl+Shift+T

Related Information

- **Migrating to Quartus Prime Pro Edition** on page 1-3
- **Recommended Design Practices** on page 10-1
- **Recommended HDL Coding Styles** on page 11-1

Synthesis Settings

You can access the following settings to customize the results of design synthesis.

Design File Options

You can add, remove, and change the compilation order of the files in the current project. You can also specify the default version for synthesis of HDL files. Click **Project** > **Add/Remove Files in Project** to locate and add or remove design files in the project. Click **Up** or **Down** to change the file compilation order.

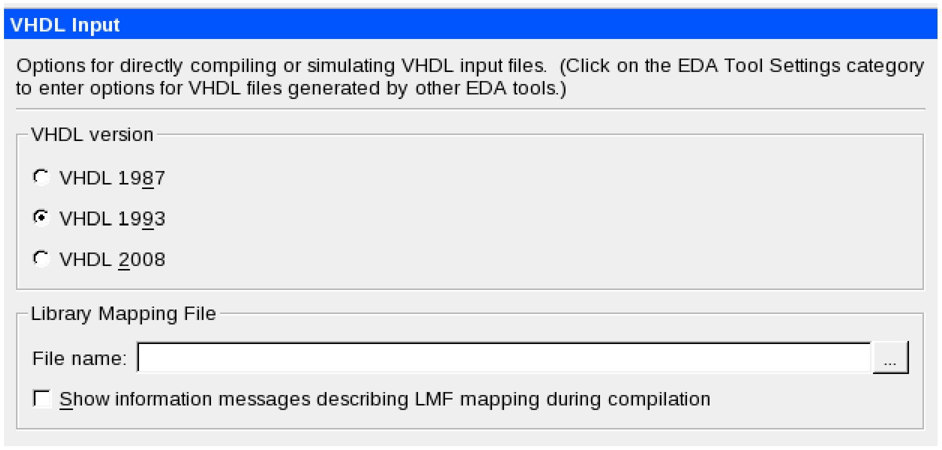
The **File Properties** dialog box, which displays the name, file type, user library information about a file, the file version, the last modification time, and allows you to change the third-party EDA tool for the file.

VHDL Input Settings (Settings Dialog Box)

Click **Assignments** > **Settings** > **VHDL Input** to specify options for synthesis of VHDL input files. Click **Up** or **Down** to change the file compilation order.

The **File Properties** dialog box, which displays the name, file type, user library information about a file, the file version, the last modification time, and allows you to change the third-party EDA tool for the file.

Figure 12-9: VHDL Input Settings


VHDL Input

Options for directly compiling or simulating VHDL input files. (Click on the EDA Tool Settings category to enter options for VHDL files generated by other EDA tools.)

VHDL version
- VHDL 1987
- [✓ VHDL 1993]
- VHDL 2008

Library Mapping File

File name:

☐ Show information messages describing LMF mapping during compilation

Table 12-5: VHDL Input Settings

Setting	Description
VHDL Version	Directs synthesis to process VDHL input design files using the specified standard. You can select any of the supported language standards to match your VHDL files.
Library Mapping File	Allows you to optionally specify an Altera-provided Library Mapping Files (**.lmf**) for use in synthesizing VHDL files that contain non Altera functions that are mapped to Altera functions. You can specify the full path name of the LMF in the **File name** box.

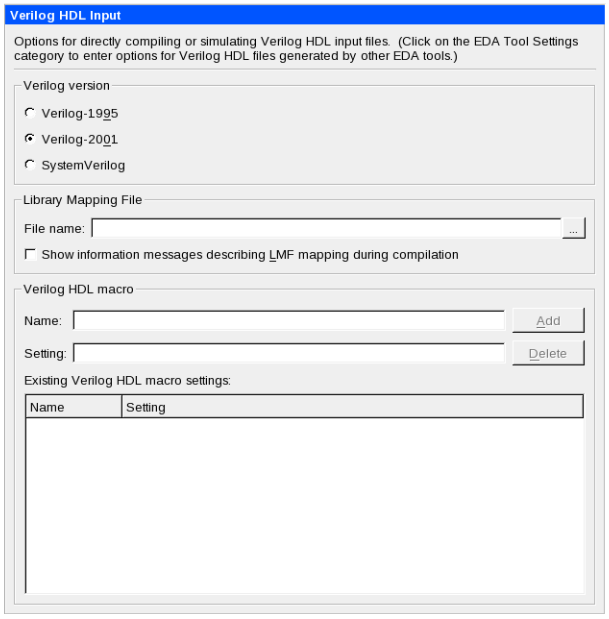
Related Information

- **VHDL Synthesis Support** on page 12-9
- **Migrating to Quartus Prime Pro Edition** on page 1-3

Verilog HDL Input Settings (Settings Dialog Box)

Click **Assignments** > **Settings** > **Verilog HDL Input** to specify options for synthesis of Verilog HDL input files. Click **Up** or **Down** to change the file compilation order.

Figure 12-10: Verilog HDL Input



Verilog HDL Input

Options for directly compiling or simulating Verilog HDL input files. (Click on the EDA Tool Settings category to enter options for Verilog HDL files generated by other EDA tools.)

Verilog version
- Verilog-1995
- [✓ Verilog-2001]
- SystemVerilog

Library Mapping File
File name:
☐ Show information messages describing LMF mapping during compilation

Verilog HDL macro
Name:
Setting:
Existing Verilog HDL macro settings:

Name	Setting

Table 12-6: Verilog HDL Input Settings

Setting	Description
Verilog Version	Directs synthesis to process Verilog HDL input design files using the specified standard. You can select any of the supported language standards to match your Verilog HDL files.
Library Mapping File	Allows you to optionally specify an Altera-provided Library Mapping Files (**.lmf**) for use in synthesizing Verilog HDL files that contain non Altera functions mapped to Altera functions. You can specify the full path name of the LMF in the **File name** box.
Verilog HDL Macro	Verilog HDL macros are pre-compiler directives which can be added to Verilog HDL files to define constants, flags, or other features by **Name** and **Setting**. Macros that you add appear in the **Existing Verilog HDLmacro settings** list.

Related Information

- **Verilog and SystemVerilog Synthesis Support** on page 12-6
- **Migrating to Quartus Prime Pro Edition** on page 1-3
- **Recommended Design Practices** on page 10-1
- **Recommended HDL Coding Styles** on page 11-1

Optimization Focus Options

You can specify options that focus synthesis optimizations to meet your specific performance, power, or logic usage goals. Click **Assignments** > **Settings** > **Compiler Settings** to access these options.

Optimization Modes

The following options direct the focus of Compiler optimization efforts during synthesis.

Table 12-7: Optimization Modes (Compiler Settings Page)

Optimization Modes	Description
Balanced (Normal Flow)	Optimizes synthesis for balanced implementation that respects timing constraints.
Performance (High effort - increases runtime)	Makes high effort to optimize synthesis for speed performance. High effort increases synthesis run time.
Performance (Aggressive - increases runtime and area)	Makes aggressive effort to optimize synthesis for speed performance. Aggressive effort increases synthesis run time and device resource use.
Power (High effort - increases runtime)	Makes high effort to optimize synthesis for low power. High effort increases synthesis run time.
Power (Aggressive - increases runtime, reduces performance)	Makes aggressive effort to optimize synthesis for low power. Aggressive effort increases synthesis time and reduces speed performance.
Area (Aggressive - reduces performance)	Makes aggressive effort to reduce the device area required to implement the design.

Prevent Register Optimizations

The following settings control general register optimization during synthesis.

Table 12-8: Prevent Register Optimizations (Compiler Settings Page)

Optimization mode	Description
Prevent register merging	Prevents automatic removal of identical registers. When this option is off, if two registers generate the same logic, one register is eliminated and the remaining register fans-out to the deleted register's destinations. This option is useful if you wish to prevent the Compiler from removing intentionally duplicate registers.
Prevent register duplication	Prevents automatic duplication of registers to improve design performance. When this option is turned off, the Compiler may duplicate a register and move a portion of its fan-out to the new node. This may improve routability and/or reduce the total routing wire required to route a net with many fan-outs.

Optimization mode	Description
Prevent register retiming	Prevents automatic retiming of registers to improve design performance. When this option is turned off, the Compiler may perform optimizations that move combinational logic across register boundaries, maintaining the overall logic of the design component but also balancing the data path delays between each register. The Compiler does not retime registers automatically unless a performance-focused optimization mode, or other specialized compiler setting is also enabled.

Default Parameters

When you create a parameterized design file, you can specify the parameters used within that file and optional default parameter values (which are used only if no parameter values are specified elsewhere). You can assign global, project-wide default values for parameters by clicking **Assignments** > **Settings** > **Default Parameter Settings** page.

In a Block Design File, you specify the parameters used within the current file with `PARAM` primitives; in HDL files, the you specify the parameters in a `Parameters Statement`; in a VHDL Design File, specify parameters in the `Generic Clause` of the `Entity Declaration`.

A parameter name can contain up to 32 characters. Once you create a parameterized design file, you can use the **Create AHDL Include Files for Current File** command, and the **Create Symbol Files for Current File** command to create default AHDL Function Prototypes in AHDL Include Files (**.inc**) and symbols in Block Symbol Files, respectively, that include the names (but not the values) of parameters used within the file. You can edit the parameters and parameter values for a Block Design File on the **Parameters** tab in the **Symbol Properties** dialog box.

These parameter names and values then appear as the defaults for each instance of the symbol when it is first entered in a Block Design File. Once you enter the symbol in a Block Design File, these default parameters and values can be customized on **Parameters** tab in the **Block Properties** dialog box on an instance-by-instance basis.

Default Parameter Options

Option	Description
Parameter	Allows you to add a new default parameter to the current project, or change an existing default parameter. Global default parameter values and Block Design File instance parameter values do not have explicitly declared types. In most cases, synthesis can correctly infer the type from the value without ambiguity. For example, "ABC" is interpreted as a string, 123 as an integer, and 15.4 as a floating-point value. In other cases, such as when the language of the instantiated subdesign is VHDL, synthesis uses the type of the parameter/generic in the instantiated entity to determine how to interpret the value, so that a value of 123 is interpreted as a string if the VHDL parameter uses the string type. 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 a Block Design File to Verilog, you can use '1 as the parameter value, and to pass a 4-bit binary vector from a Block Design File to Verilog, you can use 4'b1111 for a parameter of a Verilog entity If synthesis cannot infer the correct type of a parameter value, specify the parameter value in a type-encoded format where the first or first and second character of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string 1010 from a Block Design File to Verilog HDL, you cannot simply use the value 1001, because synthesis interprets it as a decimal value. You cannot use the string "1001", because synthesis interprets it as an ASCII string. You must use the type-encoded string B"1001" for synthesis to interpret the parameter value correctly. Altera recommends using the type-encoded format only when necessary to resolve ambiguity
Existing Parameter Settings	Displays the default parameters and their settings for the current project
Change	Allows you to add new assignments or parameters, or edit existing assignments and parameters.

Advanced Synthesis Settings

The Quartus Prime software provides advanced synthesis options that you can use in combination to meet your specific synthesis goals. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** to access these options.

Use the **Search** field to quickly locate any full or partial option name. Use the **Optimization Mode** setting to quickly select various predefined combinations of these settings tailored for specific design goals.

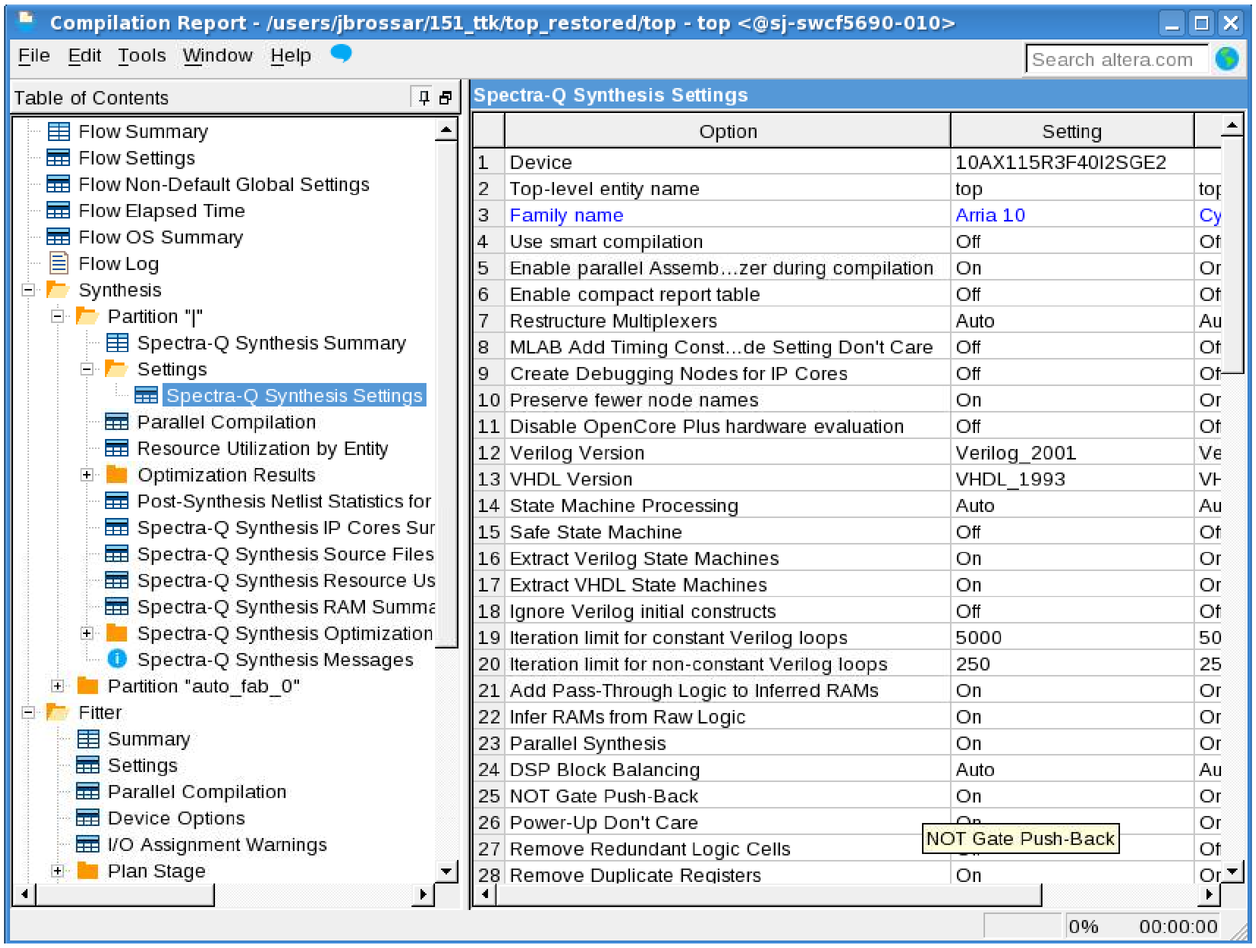
Related Information

Advanced Synthesis Settings Reference on page 12-34

Viewing Synthesis Results

The Report window displays synthesis results for each partition in the current project revision. The Compilation Report opens automatically after compilation processing. Click **Processing** > **Compilation Report** > to open the report manually.

Figure 12-11: Spectra-Q Synthesis Reports per Design Partition



Compilation Report - /users/jbrossar/151_ttk/top_restored/top - top <@sj-swcf5690-010>

File Edit Tools Window Help

Search altera.com

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Synthesis
  - Partition "|"
    - Spectra-Q Synthesis Summary
    - Settings
      - Spectra-Q Synthesis Settings
    - Parallel Compilation
    - Resource Utilization by Entity
    - Optimization Results
    - Post-Synthesis Netlist Statistics for
    - Spectra-Q Synthesis IP Cores Sur
    - Spectra-Q Synthesis Source Files
    - Spectra-Q Synthesis Resource Us
    - Spectra-Q Synthesis RAM Summa
    - Spectra-Q Synthesis Optimization
    - Spectra-Q Synthesis Messages
  - Partition "auto_fab_0"
- Fitter
  - Summary
  - Settings
  - Parallel Compilation
  - Device Options
  - I/O Assignment Warnings
  - Plan Stage

Spectra-Q Synthesis Settings

	Option	Setting	
1	Device	10AX115R3F40I2SGE2	
2	Top-level entity name	top	to
3	Family name	Arria 10	Cy
4	Use smart compilation	Off	Of
5	Enable parallel Assemb...zer during compilation	On	Or
6	Enable compact report table	Off	Of
7	Restructure Multiplexers	Auto	Au
8	MLAB Add Timing Const...de Setting Don't Care	Off	Of
9	Create Debugging Nodes for IP Cores	Off	Of
10	Preserve fewer node names	On	Or
11	Disable OpenCore Plus hardware evaluation	Off	Of
12	Verilog Version	Verilog_2001	Ve
13	VHDL Version	VHDL_1993	VH
14	State Machine Processing	Auto	Au
15	Safe State Machine	Off	Of
16	Extract Verilog State Machines	On	Or
17	Extract VHDL State Machines	On	Or
18	Ignore Verilog initial constructs	Off	Of
19	Iteration limit for constant Verilog loops	5000	50
20	Iteration limit for non-constant Verilog loops	250	25
21	Add Pass-Through Logic to Inferred RAMs	On	Or
22	Infer RAMs from Raw Logic	On	Or
23	Parallel Synthesis	On	Or
24	DSP Block Balancing	Auto	Au
25	NOT Gate Push-Back	On	Or
26	Power-Up Don't Care	On	Or
27	Remove Redundant Logic Cells		Of
28	Remove Duplicate Registers	On	Or

NOT Gate Push-Back

0% 00:00:00

Table 12-9: Synthesis Reports

Compiler Module	Function
Spectra-Q Synthesis Summary	Shows summary information about synthesis, such as the status, date, software version, entity name, device family, timing model status, and various types of logic utilization.
Spectra-Q Synthesis Settings	Lists the value of all synthesis settings during design processing.
Parallel Compilation	Lists specifications for any use of parallel processing during synthesis.
Resource Utilization By Entity	Lists the quantity of all types of logic usage for each entity in design synthesis.

Compiler Module	Function
Multiplexor Restructuring Statistics	Provides statistic for the amount of multiplexor restructuring performed during design synthesis.
Spectra-Q Synthesis IP Cores Summary	Lists details about each IP core instance in design synthesis. Details include IP core name, vendor, version, license type, entity instance, and IP include file.
Spectra-Q Synthesis Source Files Read	Lists details about all source files included in design synthesis. Details include file path, file type, and any library information.
Spectra-Q Synthesis Resource Usage Summary for Partition	Lists the quantity of all types of logic usage for each design partition design synthesis.
Spectra-Q Synthesis RAM Summary for Partition	Lists RAM usage details for each design partition design synthesis. Details include the name, type, mode, and density.
Register Statistics	Lists the number of registers using various types of global signals.
Spectra-Q Synthesis Messages	Lists all info, warning, and error messages that report conditions observed during the Analysis & Synthesis process. Right-click a message in the report and click **Help** to display on the selected message, or click **Locate** to view a list of options available for the selected message.

Running Place & Route

The Compiler's Fitter module performs design place and route. You can run the Fitter process as part of a full design compilation, or as an independent process following successful design synthesis. Prior to running the Fitter, you can specify a broad range of settings that customize place and route:

- Click **Assignments** > **Settings** > **Compiler Settings** to specify general performance, power, or logic usage goals for fitting
- Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)** to fine tune place and route with advanced Fitter options

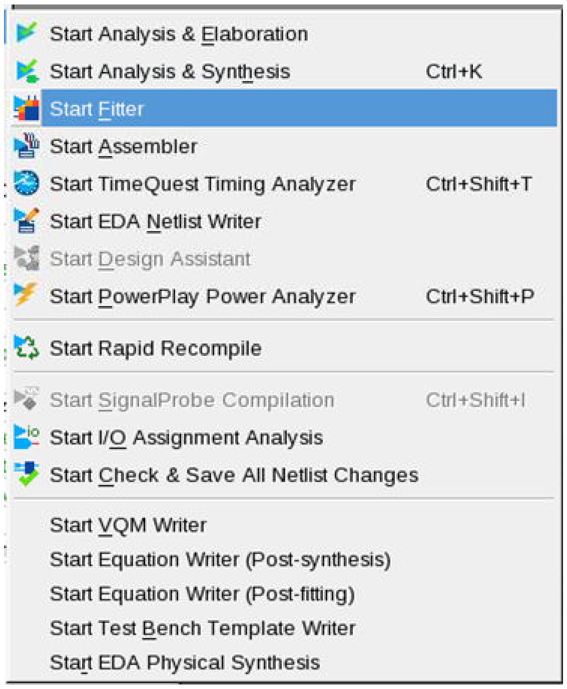
When you are ready to run place and route, click **Processing** > **Start** > **Start Fitter**. Alternatively, you can start the Fitter in batch mode at the Unix command line by running the following command:

```
quartus_fit <project name> [<options>]
```

To list all command options, type: `quartus_fit -h`.

The Compiler confirms that prerequisite modules have run, and launches the Fitter module to place and route the design logic. The Messages window dynamically displays processing information, warnings, or errors. Following Fitter processing, the **Fitter** report provides detailed information about synthesis of each design partition.

Figure 12-12: Start Place and Route (quartus_fit)

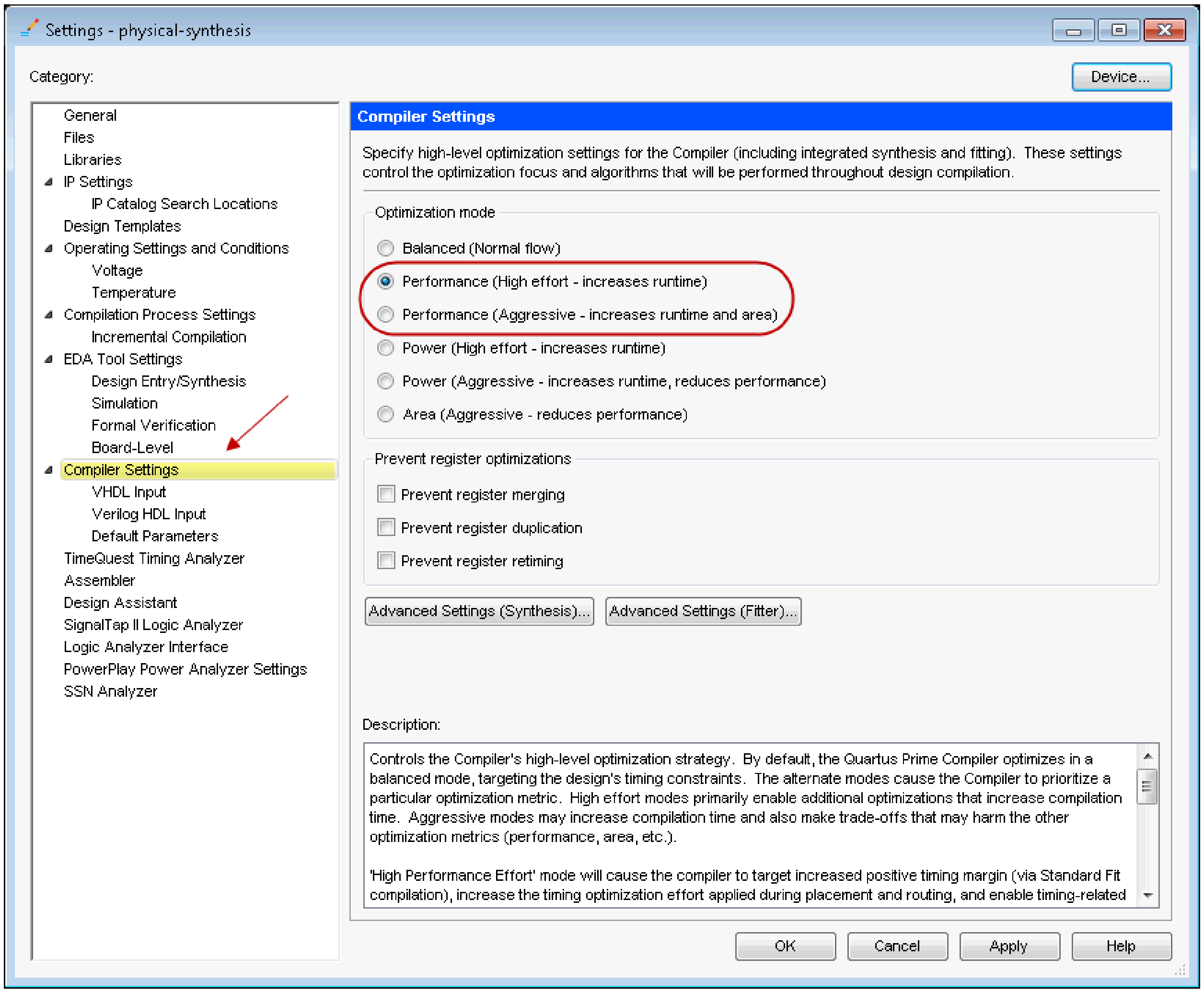

Start Analysis & Elaboration
Start Analysis & Synthesis Ctrl+K
Start Fitter
Start Assembler
Start TimeQuest Timing Analyzer Ctrl+Shift+T
Start EDA Netlist Writer
Start Design Assistant
Start PowerPlay Power Analyzer Ctrl+Shift+P
Start Rapid Recompile
Start SignalProbe Compilation Ctrl+Shift+I
Start I/O Assignment Analysis
Start Check & Save All Netlist Changes
Start VQM Writer
Start Equation Writer (Post-synthesis)
Start Equation Writer (Post-fitting)
Start Test Bench Template Writer
Start EDA Physical Synthesis

Related Information

Advanced Fitter Settings Reference on page 12-42

Setting Physical Synthesis Options

You enable **Spectra-Q Physical Synthesis** options in the Settings dialog box. When you select **Performance (High effort)** or **Performance (Aggressive)**, the **Spectra-Q Physical Synthesis** option is set to **On** automatically for supported devices. For all other optimization modes, the **Spectra-Q Physical Synthesis** option is set to **Off** by default.



Specify high-level optimization settings for the Compiler (including integrated synthesis and fitting). These settings control the optimization focus and algorithms that will be performed throughout design compilation.

Optimization mode

- ( ) Balanced (Normal flow)
- (●) Performance (High effort - increases runtime)
- ( ) Performance (Aggressive - increases runtime and area)
- ( ) Power (High effort - increases runtime)
- ( ) Power (Aggressive - increases runtime, reduces performance)
- ( ) Area (Aggressive - reduces performance)

Prevent register optimizations

- ☐ Prevent register merging
- ☐ Prevent register duplication
- ☐ Prevent register retiming

Description:

Controls the Compiler's high-level optimization strategy. By default, the Quartus Prime Compiler optimizes in a balanced mode, targeting the design's timing constraints. The alternate modes cause the Compiler to prioritize a particular optimization metric. High effort modes primarily enable additional optimizations that increase compilation time. Aggressive modes may increase compilation time and also make trade-offs that may harm the other optimization metrics (performance, area, etc.).

'High Performance Effort' mode will cause the compiler to target increased positive timing margin (via Standard Fit compilation), increase the timing optimization effort applied during placement and routing, and enable timing-related

To specify physical synthesis options, follow these steps:

1. Click **Assignments** > **Settings** > **Compiler Settings**.
2. In the **Compiler Settings**, select either high effort or aggressive performance for the **Optimization mode** to automatically enable **Spectra-Q Physical Synthesis**.
3. If you select an optimization mode other than high effort or aggressive, and you want to turn on Spectra-Q Physical Synthesis, click **Advanced Settings (Fitter)**.
4. In the **Advanced Fitter Settings** dialog box, turn on **Spectra-Q Physical Synthesis**.

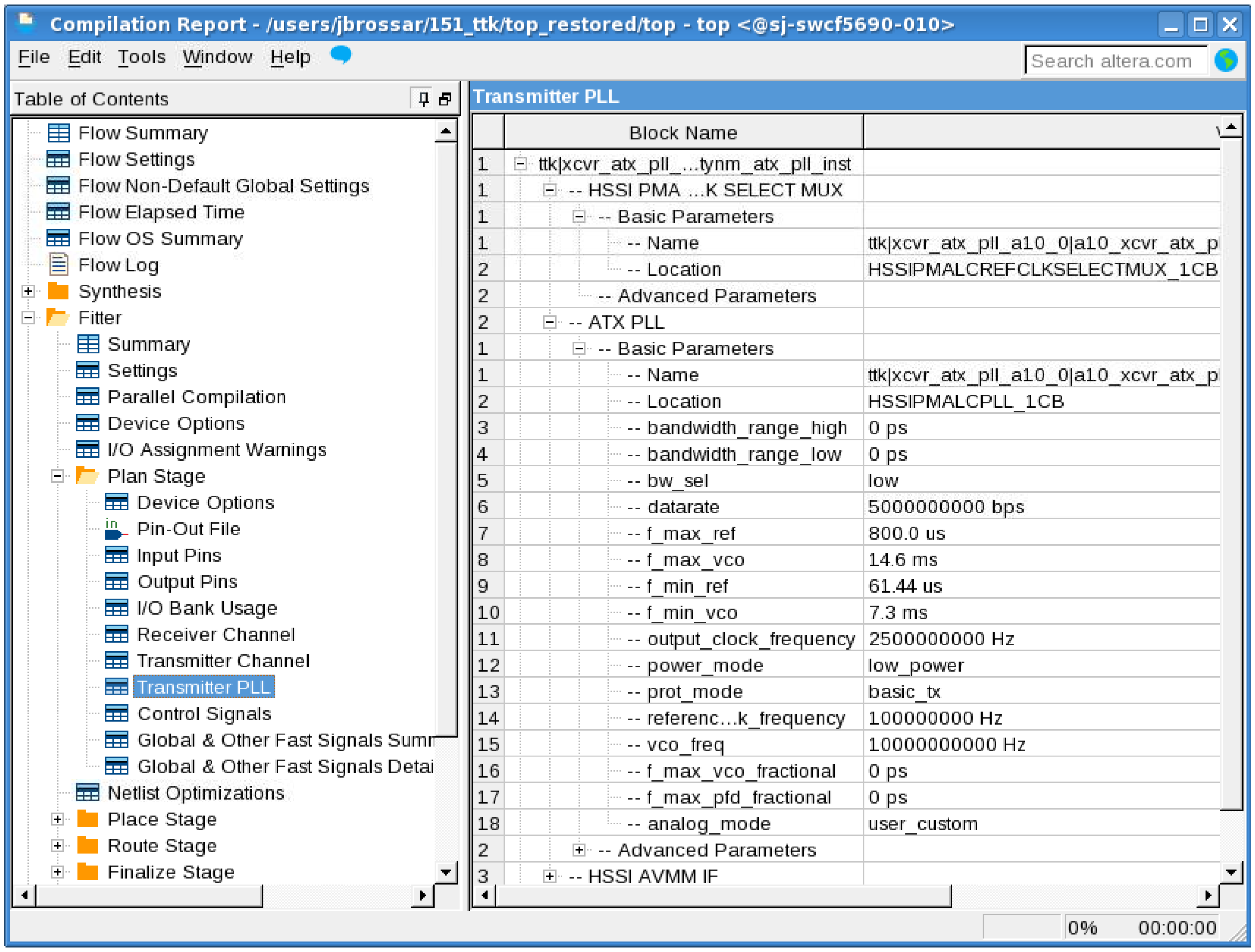
Related Information

Spectra-Q Physical Synthesis Optimization on page 12-13

Plan Stage Reports

The Report window displays Summary fitting results, as well as reports for each stage of the Fitter. The Fitter generates reports for the Plan, Place, Route, and Finalize stages.

Figure 12-13: Plan Stage Reports



Compilation Report - /users/jbrossar/151_ttk/top_restored/top - top <@sj-swcf5690-010>

File Edit Tools Window Help

Search altera.com

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Synthesis
- Fitter
  - Summary
  - Settings
  - Parallel Compilation
  - Device Options
  - I/O Assignment Warnings
  - Plan Stage
    - Device Options
    - Pin-Out File
    - Input Pins
    - Output Pins
    - I/O Bank Usage
    - Receiver Channel
    - Transmitter Channel
    - Transmitter PLL
    - Control Signals
    - Global & Other Fast Signals Sumr
    - Global & Other Fast Signals Detai
  - Netlist Optimizations
  - Place Stage
  - Route Stage
  - Finalize Stage

Transmitter PLL

	Block Name	V
1	ttk|xcvr_atx_pll_...tynm_atx_pll_inst	
1	-- HSSI PMA ...K SELECT MUX	
1	-- Basic Parameters	
1	-- Name	ttk|xcvr_atx_pll_a10_0|a10_xcvr_atx_pl
2	-- Location	HSSIPMALCREFCLKSELECTMUX_1CB
2	-- Advanced Parameters	
2	-- ATX PLL	
1	-- Basic Parameters	
1	-- Name	ttk|xcvr_atx_pll_a10_0|a10_xcvr_atx_pl
2	-- Location	HSSIPMALCPLL_1CB
3	-- bandwidth_range_high	0 ps
4	-- bandwidth_range_low	0 ps
5	-- bw_sel	low
6	-- datarate	5000000000 bps
7	-- f_max_ref	800.0 us
8	-- f_max_vco	14.6 ms
9	-- f_min_ref	61.44 us
10	-- f_min_vco	7.3 ms
11	-- output_clock_frequency	2500000000 Hz
12	-- power_mode	low_power
13	-- prot_mode	basic_tx
14	-- referenc...k_frequency	100000000 Hz
15	-- vco_freq	10000000000 Hz
16	-- f_max_vco_fractional	0 ps
17	-- f_max_pfd_fractional	0 ps
18	-- analog_mode	user_custom
2	-- Advanced Parameters	
3	-- HSSI AVMM IF	

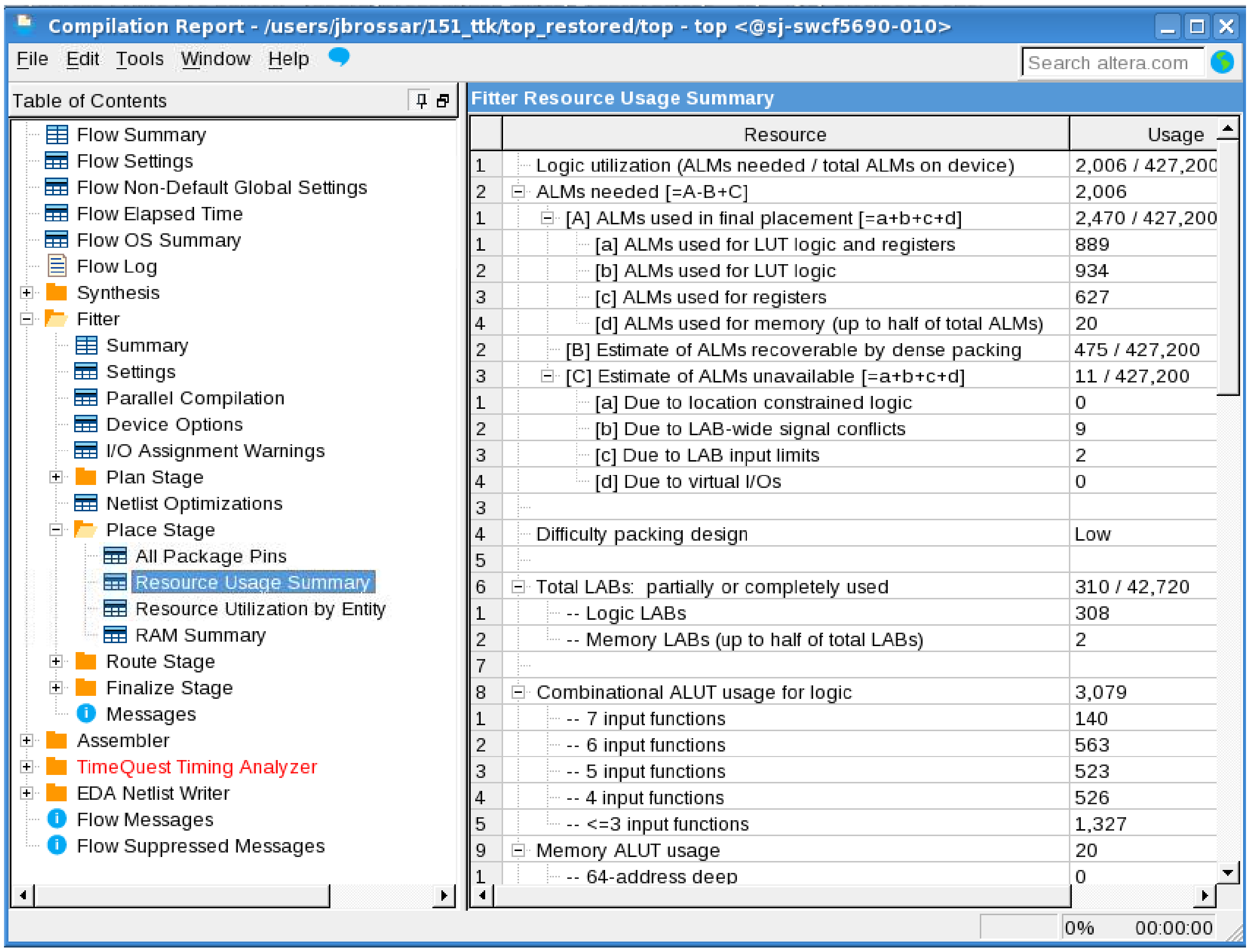
0% 00:00:00

The Compilation Report opens automatically after compilation processing, or click **Processing** > **Compilation Report** > to open the report manually. The Fitter Summary reports basic information about the Fitter run, such as the status, date, software version, entity name, device family, timing model status, and various types of logic utilization. The stage reports provide information for analysis and optimization of each Fitter stage. The Plan stage reports describes the I/O, interface, and control signals discovered during the periphery planning stage of the Fitter.

Place Stage Reports

The Place stage reports details about all device resources the Fitter allocates during logic placement. Details include the type, number, and overall percentage of each resource type.

Figure 12-14: Place Stage Reports

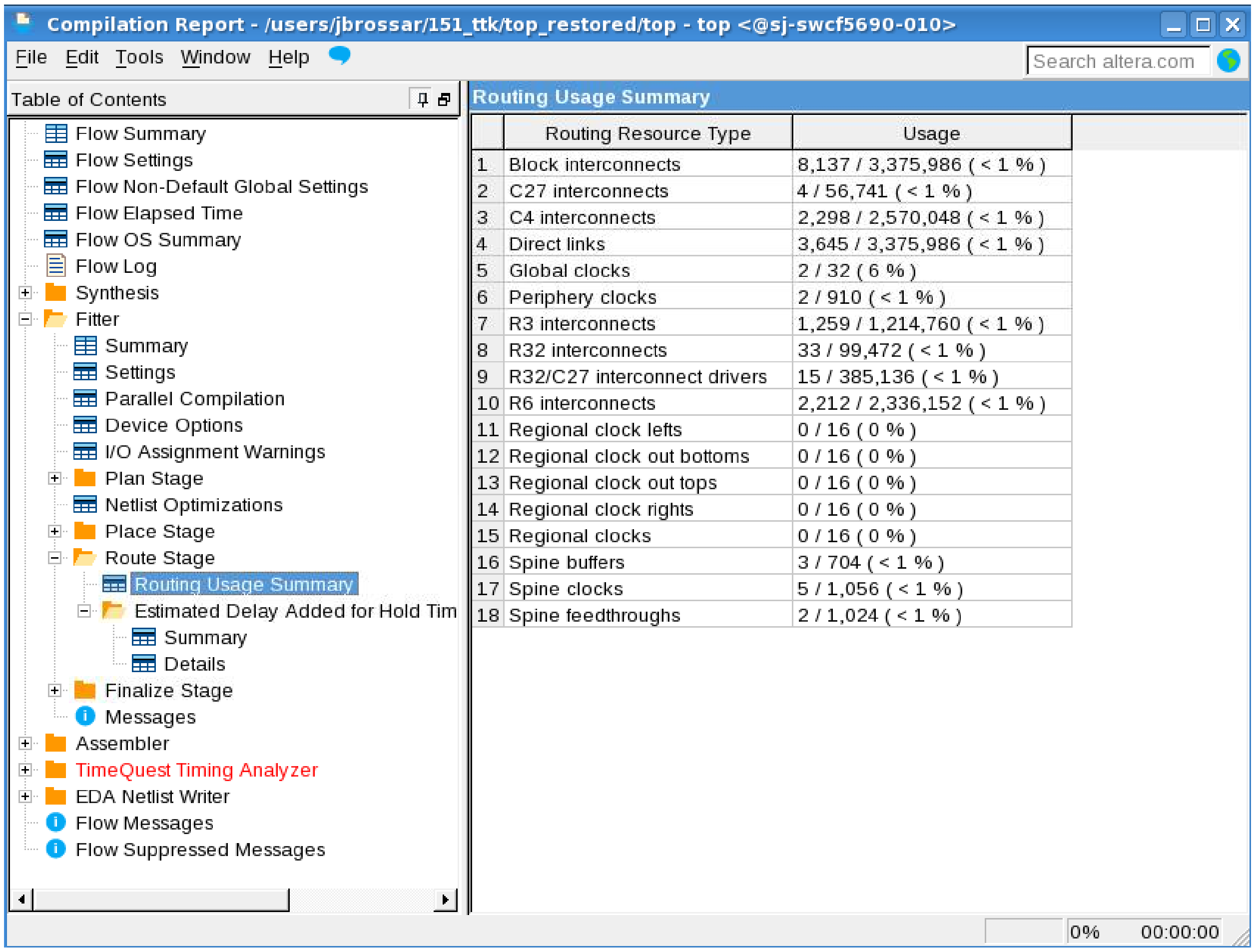


	Resource	Usage
1	Logic utilization (ALMs needed / total ALMs on device)	2,006 / 427,200
2	ALMs needed [=A-B+C]	2,006
1	[A] ALMs used in final placement [=a+b+c+d]	2,470 / 427,200
1	[a] ALMs used for LUT logic and registers	889
2	[b] ALMs used for LUT logic	934
3	[c] ALMs used for registers	627
4	[d] ALMs used for memory (up to half of total ALMs)	20
2	[B] Estimate of ALMs recoverable by dense packing	475 / 427,200
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	11 / 427,200
1	[a] Due to location constrained logic	0
2	[b] Due to LAB-wide signal conflicts	9
3	[c] Due to LAB input limits	2
4	[d] Due to virtual I/Os	0
3		
4	Difficulty packing design	Low
5		
6	Total LABs: partially or completely used	310 / 42,720
1	-- Logic LABs	308
2	-- Memory LABs (up to half of total LABs)	2
7		
8	Combinational ALUT usage for logic	3,079
1	-- 7 input functions	140
2	-- 6 input functions	563
3	-- 5 input functions	523
4	-- 4 input functions	526
5	-- <=3 input functions	1,327
9	Memory ALUT usage	20
1	-- 64-address deep	0

Route Stage Reports

The Route stage reports details about all device resources that the Fitter allocates during routing. Details include the type, number, and overall percentage of each resource type.

Figure 12-15: Route Stage Reports



Routing Usage Summary

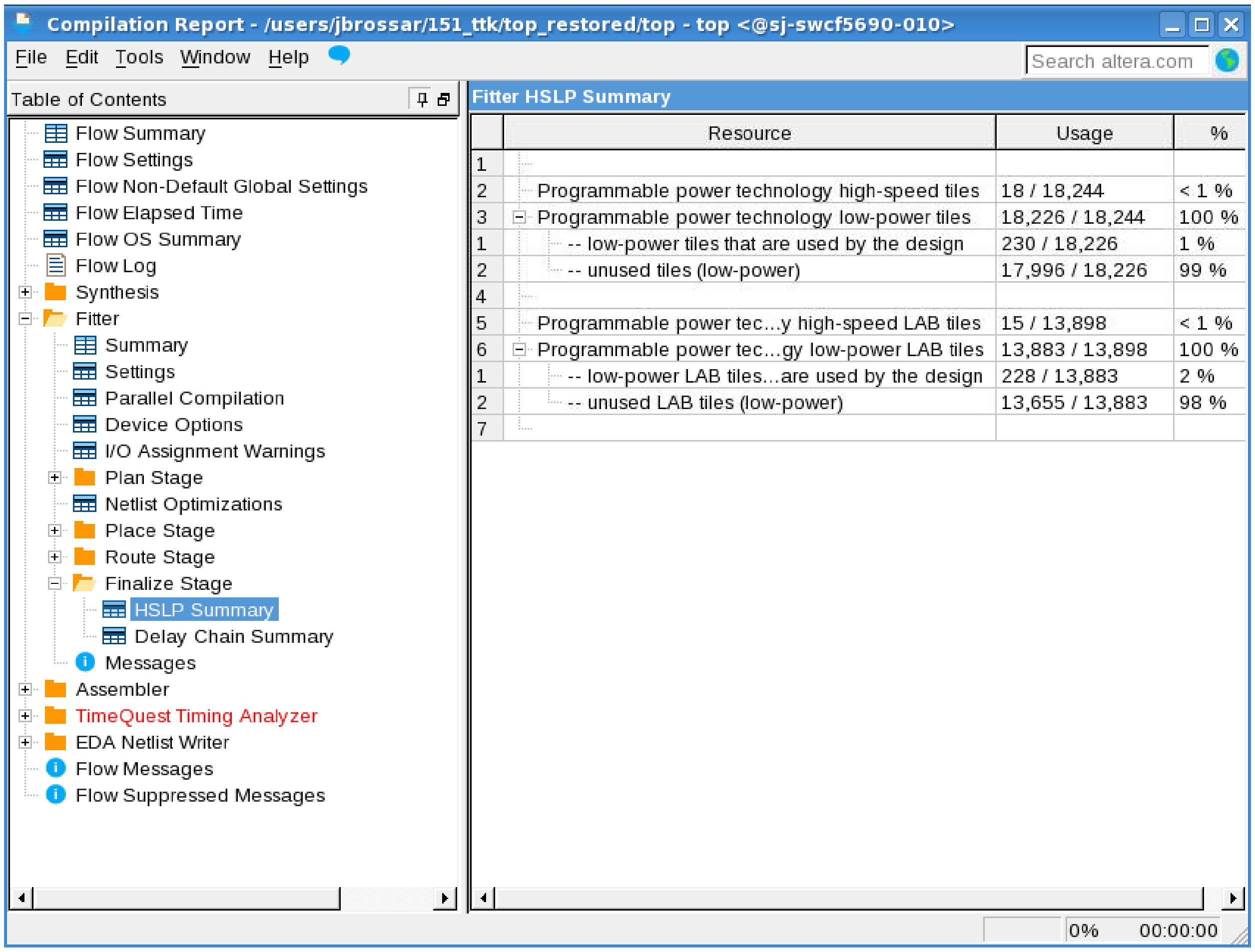
	Routing Resource Type	Usage
1	Block interconnects	8,137 / 3,375,986 (< 1 %)
2	C27 interconnects	4 / 56,741 (< 1 %)
3	C4 interconnects	2,298 / 2,570,048 (< 1 %)
4	Direct links	3,645 / 3,375,986 (< 1 %)
5	Global clocks	2 / 32 (6 %)
6	Periphery clocks	2 / 910 (< 1 %)
7	R3 interconnects	1,259 / 1,214,760 (< 1 %)
8	R32 interconnects	33 / 99,472 (< 1 %)
9	R32/C27 interconnect drivers	15 / 385,136 (< 1 %)
10	R6 interconnects	2,212 / 2,336,152 (< 1 %)
11	Regional clock lefts	0 / 16 (0 %)
12	Regional clock out bottoms	0 / 16 (0 %)
13	Regional clock out tops	0 / 16 (0 %)
14	Regional clock rights	0 / 16 (0 %)
15	Regional clocks	0 / 16 (0 %)
16	Spine buffers	3 / 704 (< 1 %)
17	Spine clocks	5 / 1,056 (< 1 %)
18	Spine feedthroughs	2 / 1,024 (< 1 %)

This report also includes the Estimated Delay Added for Hold Timing report if you enable the **Optimize Hold Timing** option in **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**.

Finalize Stage Reports

The Finalize stage reports details about final placement and routing operations, including Programmable Power Technology tile and delay chain summary information.

Figure 12-16: Finalize Stage Reports

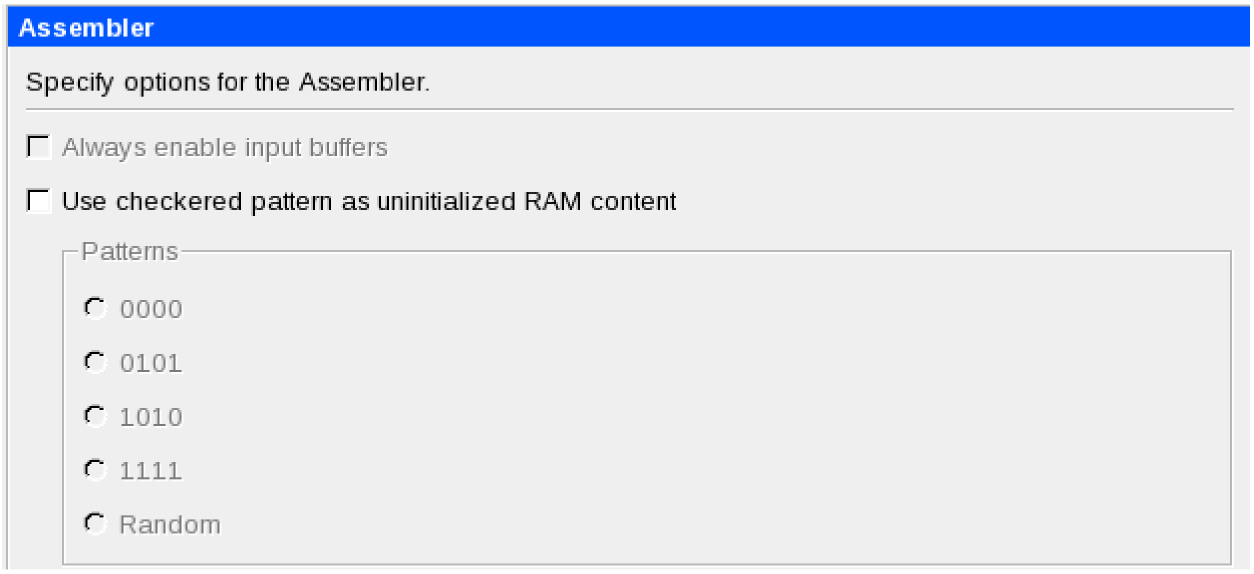


	Resource	Usage	%
1			
2	Programmable power technology high-speed tiles	18 / 18,244	< 1 %
3	Programmable power technology low-power tiles	18,226 / 18,244	100 %
1	-- low-power tiles that are used by the design	230 / 18,226	1 %
2	-- unused tiles (low-power)	17,996 / 18,226	99 %
4			
5	Programmable power tec...y high-speed LAB tiles	15 / 13,898	< 1 %
6	Programmable power tec...gy low-power LAB tiles	13,883 / 13,898	100 %
1	-- low-power LAB tiles...are used by the design	228 / 13,883	2 %
2	-- unused LAB tiles (low-power)	13,655 / 13,883	98 %
7			

Generating Programming Files

The Compiler's Assembler module generates files for device programming. You can run the Assembler process as part of a full design compilation, or as an independent process following successful design place and route. Prior to running the Assembler, you can specify settings that customize programming file generation. Click **Assignments** > **Settings** > **Assembler** to access settings that customize programming file generation.

Figure 12-17: Assembler Settings


Assembler

Specify options for the Assembler.

- [ ] Always enable input buffers
- [ ] Use checkered pattern as uninitialized RAM content

Patterns

- ○ 0000
- ○ 0101
- ○ 1010
- ○ 1111
- ○ Random

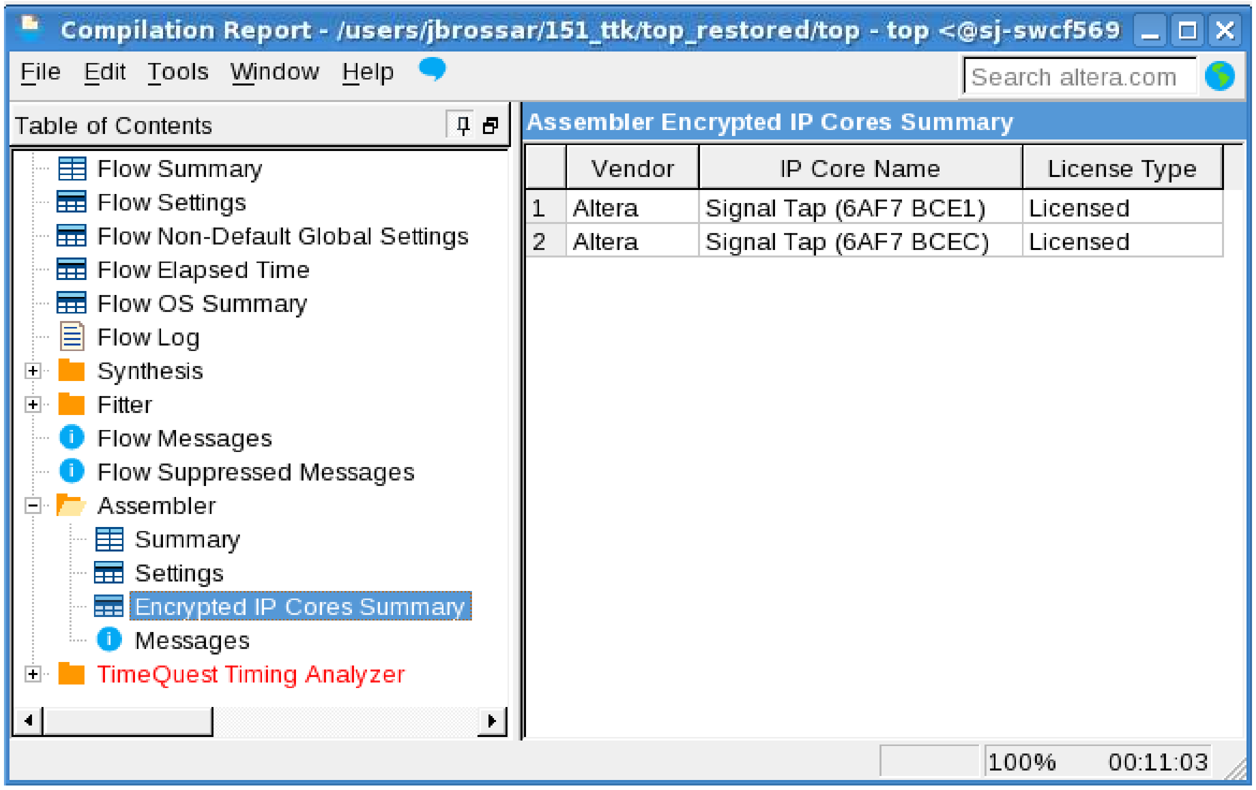
When you are ready to run place and route, click **Processing** > **Start** > **Start Assembler**. Alternatively, you can start the Assembler in batch mode at the Unix command line by running the following command:

```
quartus_asm <project name> [<options>]
```

To list all command options, type: `quartus_asm -h`.

The Compiler confirms that prerequisite modules have run, and launches the Assembler module to generate one or more programming files, according to your specifications. The Messages window dynamically displays processing information, warnings, or errors. Following Assembler processing, the **Assembler** report provides detailed information about programming files, including programming file Summary and Encrypted IP information.

Figure 12-18: Assembler Reports



Assembler Encrypted IP Cores Summary

	Vendor	IP Core Name	License Type
1	Altera	Signal Tap (6AF7 BCE1)	Licensed
2	Altera	Signal Tap (6AF7 BCEC)	Licensed

Related Information

Programming Altera Devices

Reducing Compilation Process Time

Running a full compilation including all Compiler modules on a large design can be time consuming. The Quartus Prime Pro Edition software supports the following strategies to reduce overall design compilation time:

- Rapid Recompilation of changed blocks—the Compiler reuses previous compilation results and does not reprocess unchanged design blocks
- Parallel compilation—detects and uses multiple processors to reduce compilation time (for systems with multiple processor cores)

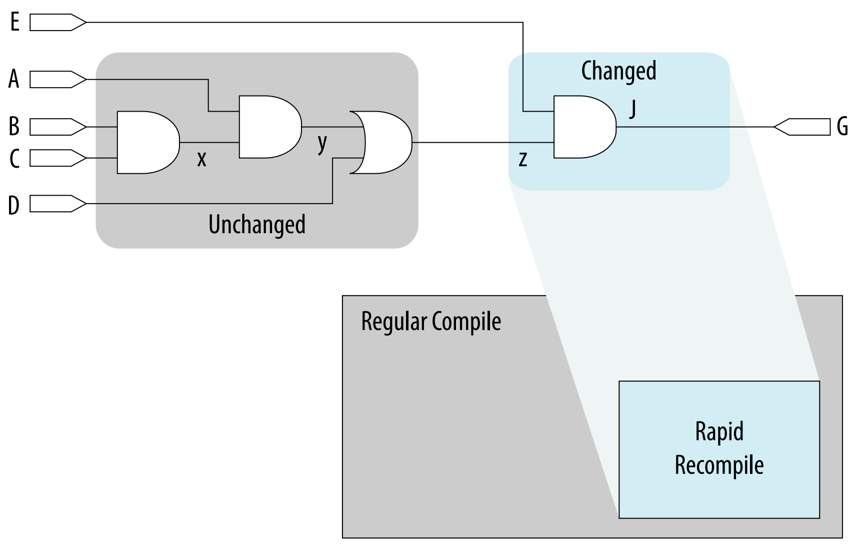
Related Information

- Using Rapid Recompile on page 12-32
- Using Parallel Compilation with Multiple Processors on page 12-33
- Reducing Compilation Time

Using Rapid Recompile

Rapid Recompile automatically reuses previous synthesis, placement, and routing results to reduce subsequent recompilation time and timing variations after making small design changes.

Figure 12-19: Rapid Recompile



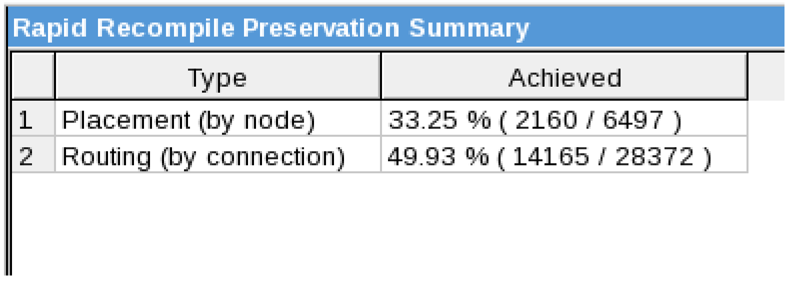
You can use Rapid Recompile to implement HDL-based functional ECO changes that affect a small subset of a large or complex design (less than 5% of total design logic), without full recompilation. Rapid Recompile can achieve up to 4x reduction in compilation time for impacted portions of the design.

To start Rapid Recompilation following an initial compilation, click **Processing** > **Start** > **Start Rapid Recompile**. Rapid Recompile implements the following type of design changes without full recompilation:

- Changes to nodes tapped by the SignalTap II Logic Analyzer
- Changes to combinational logic functions
- Changes to state machine logic (for example, new states, state transition changes)
- Changes to signal or bus latency or addition of pipeline registers
- Changes to coefficients of an adder or multiplier
- Changes register packing behavior of DSP, RAM, or I/O
- Removal of unnecessary logic
- Changes to synthesis directives

The Rapid Recompile Preservation Summary report provides detailed information about the percentage of preserved compilation results.

Figure 12-20: Rapid Recompile Preservation Summary



Rapid Recompile Preservation Summary

	Type	Achieved
1	Placement (by node)	33.25 % (2160 / 6497)
2	Routing (by connection)	49.93 % (14165 / 28372)

Using Parallel Compilation with Multiple Processors

The Quartus Prime software can detect the number of processors available on a computer and use multiple processors to reduce compilation time.

You can control the number of processors used during a compilation on a per user basis. The Quartus Prime software can use up to 16 processors to run algorithms in parallel and reduce compilation time. The Quartus Prime software turns on parallel compilation by default to enable the software to detect available multiple processors. You can specify the maximum number of processors that the software can use if you want to reserve some of the available processors for other tasks.

Note: Do not consider processors with Intel Hyper-Threading as more than one processor. If you have a single processor with Intel Hyper-Threading enabled, you should set the number of processors to one. Do not use the Intel Hyper-Threading feature for Quartus Prime compilations, because it can increase run times.

The Quartus Prime software does not necessarily use all the processors that you specify during a given compilation. Additionally, the software never uses more than the specified number of processors, enabling you to work on other tasks on your computer without it becoming slow or less responsive.

You can reduce the compilation time by up to 10% on systems with two processing cores and by up to 20% on systems with four cores. With certain design flows in which timing analysis runs alone, multiple processors can reduce the time required for timing analysis by an average of 10% when using two processors. This reduction can reach an average of 15% when using four processors.

You can also set the number of processors available for Quartus Prime compilation using the following Tcl command in your script:

```
set_global_assignment -name NUM_PARALLEL_PROCESSORS <value>
```

In this case, <*value*> is an integer from `1` to `16`.

If you want the Quartus Prime software to detect the number of processors and use all the processors for the compilation, include the following Tcl command in your script:

```
set_global_assignment -name NUM_PARALLEL_PROCESSORS ALL
```

The use of multiple processors does not affect the quality of the fit. For a given Fitter seed on a specific design, the fit is exactly the same, regardless of whether the Quartus Prime software uses one processor or multiple processors. The only difference between compilations using a different number of processors is the compilation time.

The Parallel Compilation report provides detailed information about compilation using multiple processors.

Figure 12-21: Parallel Compilation Report

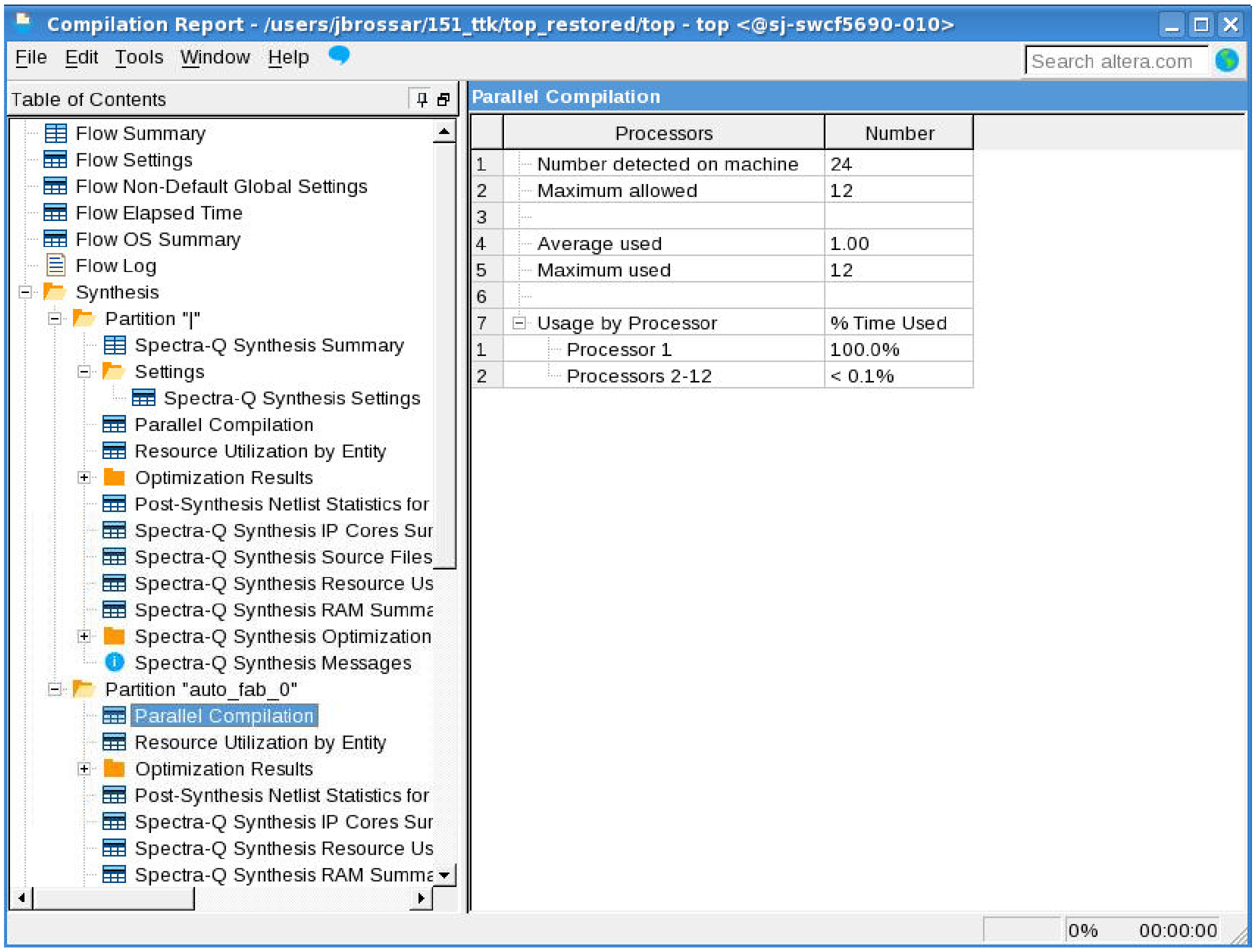

Compilation Report - /users/jbrossar/151_ttk/top_restored/top - top <@sj-swcf5690-010>

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Synthesis
  - Partition "|"
    - Spectra-Q Synthesis Summary
    - Settings
      - Spectra-Q Synthesis Settings
    - Parallel Compilation
    - Resource Utilization by Entity
    - Optimization Results
    - Post-Synthesis Netlist Statistics for
    - Spectra-Q Synthesis IP Cores Sur
    - Spectra-Q Synthesis Source Files
    - Spectra-Q Synthesis Resource Us
    - Spectra-Q Synthesis RAM Summa
    - Spectra-Q Synthesis Optimization
    - Spectra-Q Synthesis Messages
  - Partition "auto_fab_0"
    - Parallel Compilation
    - Resource Utilization by Entity
    - Optimization Results
    - Post-Synthesis Netlist Statistics for
    - Spectra-Q Synthesis IP Cores Sur
    - Spectra-Q Synthesis Resource Us
    - Spectra-Q Synthesis RAM Summa

Parallel Compilation

	Processors	Number
1	Number detected on machine	24
2	Maximum allowed	12
3		
4	Average used	1.00
5	Maximum used	12
6		
7	Usage by Processor	% Time Used
1	Processor 1	100.0%
2	Processors 2-12	< 0.1%

Related Information

- Processing Page (Options Dialog Box)
- Compilation Process Settings Page (Settings Dialog Box)
  For more information about how to control the number of processors used during compilation for a specific project, refer to Quartus Prime Help.

Advanced Synthesis Settings Reference

The following is a quick reference of all Advanced Synthesis Settings. Click **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Synthesis)** to access these options.

Table 12-10: Advanced Synthesis Settings (1 of 13)

Option	Description
Allow Any RAM Size for Recognition	Allows the Compiler to infer RAMs of any size, even if they don't meet the current minimum requirements.

Option	Description
Allow Any ROM Size for Recognition	Allows the Compiler to infer ROMs of any size even if the ROMs do not meet the design's current minimum size requirements.
Allow Any Shift Register Size for Recognition	Allows the Compiler to infer shift registers of any size even if they do not meet the design's current minimum size requirements.
Allow Any Shift Register Merging Across Hierarchies	Allows the Compiler to take shift registers from different hierarchies of the design and put them in the same RAM.
Allow Synchronous Control Signals	Allows the Compiler to utilize synchronous clear and/or synchronous load signals in normal mode logic cells. Turning on this option helps to reduce the total number of logic cells used in the design, but might negatively impact the fitting since synchronous control signals are shared by all the logic cells in a LAB.

Table 12-11: Advanced Synthesis Settings (2 of 13)

Option	Description
Analysis & Synthesis Message Level	Specifies the type of Analysis & Synthesis messages displayed. **Low** displays only the most important Analysis & Synthesis messages. **Medium** displays most messages, but hides the detailed messages. High displays all messages.
Auto Carry Chains	Allows the Compiler to create carry chains automatically by inserting CARRY_SUM buffers into the design. The length of the chains is controlled with the **Carry Chain Length** option. If this option is turned off, CARRY buffers are ignored, but CARRY_SUM buffers are unaffected. The **Auto Carry Chains** option is ignored if you select **Product Term** or **ROM** as the setting for the **Technology Mapper** option.
Auto Clock Enable Replacement	Allows the Compiler to find logic that feeds a register and move the logic to the register's clock enable input port.
Auto DSP Block Replace-ment	Allows the Compiler to find a multiply-accumulate function or a multiply-add function that can be replaced with the altmult_accum or the altmult_add IP core.
Auto Gated Clock Conversion	Automatically converts gated clocks to use clock enable pins. Clock gating logic can contain AND, OR, MUX, and NOT gates. Turning on this option may increase memory use and overall run time. You must use the TimeQuest Timing Analyzer for timing analysis, and you must define all base clocks in Synopsys Design Constraints (SDC) format.

Table 12-12: Advanced Synthesis Settings (3 of 13)

Option	Description
Auto Open-Drain Pins	Allows the Compiler to automatically convert a tri-state buffer with a strong low data input into the equivalent open-drain buffer.
Auto RAM Replacement	Allows the Compiler to find a set of registers and logic that can be replaced with the altsyncram or the lpm_ram_dp IP core. Turning on this option may change the functionality of the design.
Auto ROM Replacement	Allows the Compiler to find logic that can be replaced with the altsyncram or the lpm_rom IP core. Turning on this option may change the power-up state of the design.
Auto Resource Sharing	Allows the Compiler to share hardware resources among many similar, but mutually exclusive, operations in your HDL source code. If you enable this option, the Compiler merges compatible addition, subtraction, and multiplication operations. By merging operations, this may reduce the area required by your design. Because resource sharing introduces extra muxing and control logic on each shared resource, it may negatively impact the final fMAX of your design.
Auto Shift Register Placement	Allows the Compiler to find a group of shift registers of the same length that can be replaced with the altshift_taps IP core. The shift registers must all use the same clock and clock enable signals, must not have any other secondary signals, and must have equally spaced taps that are at least three registers apart.

Table 12-13: Advanced Synthesis Settings (4 of 13)

Option	Description
Block Design Naming	Specify the naming scheme used for the block design. This option is ignored if it is assigned to anything other than a design entity.
Carry Chain Length	Specifies the maximum allowable length of a chain of both user-entered and Compiler-synthesized CARRY_SUM buffers. Carry chains that exceed this length are broken into separate chains.
Clock MUX Protection	Causes the multiplexers in the clock network to be decomposed to 2to1 multiplexer trees, and protected from being merged with, or transferred to, other logic. This option helps the TimeQuest Timing Analyzer to understand clock behavior.

Option	Description
Create Debugging Nodes for IP Cores	Make certain nodes (for example, important registers, pins, and state machines) visible for all the IP cores in a design. You can use IP core nodes to effectively debug the IP core, particularly when using the IP core with the SignalTap II Logic Analyzer. The Node Finder, using SignalTap II Logic Analyzer filters, displays all the nodes that Analysis & Synthesis makes visible. When making the debugging nodes visible, Analysis & Synthesis can change the f_{MAX} and number of logic cells in IP cores.
DSP Block Balancing	Allows you to control the conversion of certain DSP block slices during DSP block balancing.

Table 12-14: Advanced Synthesis Settings (5 of 13)

Option	Description
Disable DSP Negate Inferencing	Allows you to specify whether to use the negate port on an inferred DSP block.
Force Use of Synchronous Clear Signals	Forces the Compiler to utilize synchronous clear signals in normal mode logic cells. Turning on this option helps to reduce the total number of logic cells used in the design, but might negatively impact the fitting since synchronous control signals are shared by all the logic cells in a LAB.
HDL Message Level	Specifies the type of HDL messages you want to view, including messages that display processing errors in the HDL source code. **Level1** displays only the most important HDL messages. **Level2** displays most HDL messages, including warning and information based messages. **Level3** displays all HDL messages, including warning and information based messages and alerts about potential design problems or lint errors.
Ignore CARRY Buffers	Ignores CARRY_SUM buffers in the design. This option is ignored if it is applied to anything other than an individual CARRY_SUM buffer or to a design entity containing CARRY_SUM buffers.
Ignore CASCADE Buffers	Ignores CASCADE buffers that are instantiated in the design. This option is ignored if it is applied to anything other than an individual CASCADE buffer or a design entity containing CASCADE buffers.

Table 12-15: Advanced Synthesis Settings (6 of 13)

Option	Description
Ignore GLOBAL Buffers	Ignores GLOBAL buffers that are instantiated in the design. This option is ignored if it is applied to anything other than an individual GLOBAL buffer or a design entity containing GLOBAL buffers.

Option	Description
Ignore LCELL Buffers	Ignores LCELL buffers that are instantiated in the design. This option is ignored if it is applied to anything other than an individual LCELL buffer or a design entity containing LCELL buffers.
Ignore Maximum Fan-Out Assignments	Directs the Compiler to ignore the Maximum Fan-Out Assignments on a node, an entity, or the whole design.
Ignore ROW GLOBAL Buffers	Ignores ROW GLOBAL buffers that are instantiated in the design. This option is ignored if it is applied to anything other than an individual GLOBAL buffer or a design entity containing GLOBAL buffers.
Ignore SOFT Buffers	Ignores SOFT buffers that are instantiated in the design. This option is ignored if it is applied to anything other than an individual SOFT buffer or a design entity containing SOFT buffers.

Table 12-16: Advanced Synthesis Settings (7 of 13)

Option	Description
Ignore Verilog Initial Constructs	Instructs Analysis & Synthesis to ignore initial constructs and variable declaration assignments in your Verilog HDL design files. By default, Analysis & Synthesis derives power-up conditions for your design by elaborating these constructs. This option is provided for backwards compatibility with previous versions of the Quartus Prime software that ignored these constructs by default. You can use this option to restore the previous behavior of your design in the current version of the software.
Ignore translate_off and synthesis_off Directives	Instructs Analysis & Synthesis to ignore all translate_off/synthesis_off synthesis directives in your Verilog HDL and VHDL design files. You can use this option to disable these synthesis directives and include previously ignored code during elaboration.
Infer RAMS from Raw Logic	Instructs the Compiler to infer RAM from registers and multiplexers. Some HDL patterns that differ from Altera RAM templates are initially converted into logic. However, these structures function as RAM and, because of that, the Compiler may create an altsyncram IP core instance for them at a later stage when this assignment is on. With this assignment is turned on, the Compiler may use more device RAM resources and less LABs.
Iteration Limit for Constant Verilog Loops	Defines the iteration limit for Verilog loops with loop conditions that evaluate to compile-time constants on each loop iteration. This limit exists primarily to identify potential infinite loops before they exhaust memory or trap the software in an actual infinite loop.
Iteration Limit for non-Constant Verilog Loops	Defines the iteration limit for Verilog HDL loops with loop conditions that do not evaluate to compile-time constants on each loop iteration. This limit exists primarily to identify potential infinite loops before they exhaust memory or trap the software in an actual infinite loop.

Table 12-17: Advanced Synthesis Settings (8 of 13)

Option	Description
Limit AHDL integers to 32 Bits	Specifies whether an AHDL-based design should have a limit on integer size of 32 bits. This option is provided for backward compatibility with pre-2000.09 releases of the Quartus software, which do not support integers larger than 32 bits in AHDL.
Maximum DSP Block Usage	Specifies the maximum number of DSP blocks that the DSP block balancer assumes exist in the current device for each partition. This option overrides the usual method of using the maximum number of DSP blocks the current device supports.
Maximum Number of LABs	Specifies the maximum number of LABs that Analysis & Synthesis should try to utilize for a device. This option overrides the usual method of using the maximum number of LABs the current device supports, when the value is non-negative and is less than the maximum number of LABs available on the current device.
Maximum Number of M-RAM/M144K Memory Blocks	Specifies the maximum number of M-RAM/M144K memory blocks that the Compiler may utilize for a device. This option overrides the usual method of using the maximum number of M-RAM/M144K memory blocks the selected device supports, when the value is non-negative and is less than the maximum number of M-RAM/M144K memory blocks available on the current device.
Maximum Number of M4K/M9K/M20K/M10K Memory Blocks	Specifies the maximum number of M4K,M9K,M20K,or M10K memory blocks that the Compiler may use for a device. This option overrides the usual method of using the maximum number of M4K,M9K,M20K, or M10K memory blocks the current device supports, when the value is non-negative and is less than the maximum number of M4K,M9K,M20K, or M10K memory blocks available on the current device.

Table 12-18: Advanced Synthesis Settings (9 of 13)

Option	Description
Maximum Number of Registers Created from Uninferred RAMs	Specifies the maximum number of registers that Analysis & Synthesis can use for conversion of uninferred RAMs. You can use this option as a project-wide option or on a specific partition by setting the assignment on the instance name of the partition root. The assignment on a partition overrides the global assignment (if any) for that particular partition. This option prevents synthesis from causing long compilations and running out of memory when many registers are used for uninferred RAMs. Instead of continuing the compilation, the Quartus Prime software issues an error and exits.

Option	Description
NOT Gate Push-Back	Allows the Compiler to push an inversion (that is, a NOT gate) back through a register and implement it on that register's data input if it is necessary to implement the design. If this option is turned on, a register may power up to an active-high state, so it may need to be explicitly cleared during initial operation of the device. This option is ignored if it is applied to anything other than an individual register or a design entity containing registers. If it is applied to an output pin that is directly fed by a register, it is automatically transferred to that register.
Number of Inverted Registers Reported in Synthesis Report	Specifies the maximum number of inverted registers that the Synthesis Report should display.
Number of Removed Registers Reported in Synthesis Report	Allows you to specify the maximum number of removed registers that the Synthesis Report should display.
Optimization Technique	Specifies the overall optimization goal for Analysis & Synthesis: attempt to maximize performance, minimize logic usage, or balance high performance with minimal logic usage.

Table 12-19: Advanced Synthesis Settings (10 of 13)

Option	Description
Parallel Synthesis	Enables parallel synthesis.
Perform WYSIWYG Primitive Resynthesis	Specifies whether to perform WYSIWYG primitive resynthesis during synthesis. This option uses the setting specified in the **Optimization Technique** logic option.
Power-Up Don't Care	Causes registers that do not have a **Power-Up Level logic** option setting to power up with a don't care logic level (X). When the **Power-Up Don't Care** option is turned on, the Compiler determines when it is beneficial to change the power-up level of a register to minimize the area of the design. A power-up state of zero is maintained unless there is an immediate area advantage.
PowerPlay Power Optimization During Synthesis	Controls the power-driven compilation setting of Analysis & Synthesis. This option determines how aggressively Analysis & Synthesis optimizes the design for power. **Off** does not perform any power optimizations. **Normal compilation** performs power optimizations as long as they are not expected to reduce design performance. **Extra effort** performs additional power optimizations which may reduce design performance.

Table 12-20: Advanced Synthesis Settings (11 of 13)

Option	Description
Remove Duplicate Registers	Removes a register if it is identical to another register. If two registers generate the same logic, the second one is deleted and the first one is made to fan out to the second one's destinations. Also, if the deleted register has different logic option assignments, they are ignored. This option is useful if you wish to prevent the Compiler from removing duplicate registers that you have used deliberately. You can do this by setting the option to **Off**. This option is ignored if it is applied to anything other than an individual register or a design entity containing registers.
Remove Redundant Logic Cells	Removes redundant LCELL primitives or WYSIWYG primitives. Turning this option on optimizes a circuit for area and speed. This option is ignored if it is applied to anything other than a design entity.
Report Connectivity Checks	Specifies whether the synthesis report should include the panels in the Connectivity Checks folder.
Report Parameter Settings	Specifies whether the synthesis report should include the panels in the Parameter Settings by Entity Instance folder.
Report Source Assignments	Specifies whether the synthesis report should include the panels in the Source Assignments folder.

Table 12-21: Advanced Synthesis Settings (12 of 13)

Option	Description
Resource Aware Inference for Block RAM	Specifies whether RAM, ROM, and shift-register inference should take the design and device resources into account.
Restructure Multiplexers	Reduces the number of logic elements required to implement multiplexers in a design. This option is useful if your design contains buses of fragmented multiplexers. This option repacks multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of logic elements. **On** minimizes your design area; but may negatively affect design clock speed (f_{MAX}). **Off** disables multiplexer restructuring; it does not decrease logic element usage and does not affect design clock speed (f_{MAX}). You may select **Auto** allows the Quartus Prime software to determine whether multiplexer restructuring should be enabled. The Quartus Prime software uses other synthesis settings, for example, the **Optimization Technique** option, to determine if multiplexer restructuring should be applied to the design; the **Auto** setting will decrease logic element usage but may negatively affect design clock speed (f_{MAX}).
SDC Constraint Protection	Verifies SDC constraints in register merging. This option helps to maintain the validity of SDC constraints through compilation.

Option	Description
Safe State Machine	Tells the Compiler to implement state machines that can recover from an illegal state.
Shift Register Replacement	Allows the Compiler to find a group of shift registers of the same length that can be replaced with the altshift_taps IP core. The shift registers must all use the same `aclr` signals, must not have any other secondary signals, and must have equally spaced taps that are at least three registers apart. To use this option, you must turn on the **Auto Shift Register Replacement** logic option.

Table 12-22: Advanced Synthesis Settings (13 of 13)

Option	Description
State Machine Processing	Specifies the processing style used to compile a state machine. You can use your own **User-Encoded** style, or select **One-Hot**, **Minimal Bits**, **Gray**, **Johnson**, **Sequential,** or **Auto** (Compiler-selected) encoding.
Strict RAM Replacement	When this option is **On**, the Compiler is only allowed to replace RAM if the hardware matches the design exactly.
Synchronization Register Chain Length	Specifies the maximum number of registers in a row considered as a synchronization chain. Synchronization chains are sequences of registers with the same clock, no fanout in between, such that the first register is fed by a pin, or by logic in another clock domain. These registers are considered for metastability analysis (available for some device families), and are also protected from optimizations such as retiming. When gate-level retiming is turned on, these registers are not removed. The default length is set to two.
Synthesis Effort	Controls the synthesis trade-off between compilation speed and performance and area. The default is **Auto**. You can select **Fast** for faster compilation speed at the cost of performance and area.
Timing-Driven Synthesis	Allows synthesis to use timing information during synthesis to better optimize the design.
Use LogicLock Constraints during Resource Balancing	Directs the compiler to use LogicLock constraints during DSP and RAM balancing.

Related Information

Advanced Synthesis Settings on page 12-21

Advanced Fitter Settings Reference

The following is a quick reference of all Advanced Fitter Settings. **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)** to fine tune place and route with advanced Fitter settings.

Table 12-23: Advanced Fitter Settings (1 of 8)

Option	Description
ALM Register Packing Effort	Guides aggressiveness of the Fitter in packing ALMs during register placement. Use this option to increase secondary register locations. Increasing ALM packing density may lower the number of ALMs needed to fit the design, but it may also reduce routing flexibility and timing performance. • **Low**—The Fitter avoids ALM packing configurations that combine LUTs and registers which have no direct connectivity. Avoiding these configurations may improve timing performance but increases the number of ALMs to implement the design. • **Medium**—The Fitter allows some configurations that combine unconnected LUTs and registers to be implemented in ALM locations. The Fitter makes more usage of secondary register locations within the ALM. • **High**—The Fitter enables all legal and desired ALM packing configurations. In dense designs, the Fitter automatically increases the ALM register packing effort as required to enable the design to fit.
Advanced Physical Optimization	Enables Advanced Physical Optimization to improve the quality of results with more consistent timing closure.
Allow Delay Chains	Allows the Fitter to choose the optimal delay chain to meet t_{SU} and t_{CO} timing requirements for all I/O elements. Turning on this option may reduce the number of $_{SU}$ violations while introducing a minimal number of t_H violations. Turning on this option does not override delay chain settings on individual nodes.
Allow Delay Chains for High Fanout Pins	Allows the Fitter to choose how to optimize the delay chains for high fanout input pins. You must enable **Auto Delay Chains** to enable this option. Enabling this option may reduce the number of $_{SU}$ violation, but the compile time increases significantly, as the Fitter tries to optimize the settings for all fanouts.
Auto Fit Effort Desired Slack Margin	Specifies the default worst-case slack margin the Fitter maintains for **Auto Fit** for **Fitter Effort**. If the design is likely to have at least this much slack on every path, the Fitter reduces optimization effort to reduce compilation time. Otherwise, its behavior is the same as **Standard Fit**.

Table 12-24: Advanced Fitter Settings (2 of 8)

Option	Description
Auto Global Clock	Allows the Compiler to choose the signal that feeds the most clock inputs to flipflops as a global clock signal that is made available throughout the device on the global routing paths. If you want to prevent the Compiler from automatically selecting a particular signal as global clock, set the **Global Signal** option to **Off** on that signal.

Option	Description
Auto Global Register Control Signals	Allows the Compiler to choose the signals that feed the most control signal inputs to flipflops (excluding clock signals) as global signals that are made available throughout the device on the global routing paths. Depending on the target device family, these control signals can include asynchronous clear and load, synchronous clear and load, clock enable, and preset signals. If you want to prevent the Compiler from automatically selecting a particular signal as global register control signal, set the **Global Signal** option to **Off** on that signal.
Auto Packed Registers	Allows the Compiler to combine a register and a combinational function, or to implement registers using I/O cells, RAM blocks, or DSP blocks instead of logic cells. This option controls how aggressively the Fitter combines registers with other function blocks to reduce the area of the design. Generally, the **Auto** or **Sparse Auto** settings are appropriate. The other options limit the flexibility of the Fitter to combine registers with other function blocks and can result in no fits. When **Auto**, the Fitter attempts to achieve the best performance with good area. If necessary, the Fitter combines additional logic to reduce the area of the design to within the current device. When this setting is **Sparse Auto**, the Fitter attempts to achieve the highest performance with possibly increased area, but without exceeding the logic capacity of the device. If this option is set to **Off**, the Fitter does not combine registers with other functions. The **Off** setting severely increases the area of the design and may cause a no fit. If this option is set to **Sparse**, the Fitter combines functions in a way which improves performance for many designs. If this option is set to **Normal**, the Fitter combines functions that are expected to maximize design performance and reduce area. When this option is set to **Minimize Area**, the Fitter aggressively combines unrelated functions to reduce the area required for placing the design, at the expense of performance. When this option is set to **Minimize Area with Chains**, the Fitter even more aggressively combines functions that are part of register cascade chains or can be converted to register cascade chains. If this option is set to any value but **Off**, registers combine with I/O cells to improve I/O timing (as long as the **Optimize IOC Register Placement For Timing** option allows it), and with DSP blocks and RAM blocks to reduce the area required for placing the design or to improve timing when possible.
Auto RAM to MLAB Conversion	Specifies whether the Fitter is able to convert RAMs to use LAB locations when those RAMs use **Auto** as the selected block type. If this option is changed to **Off,** then only MLAB cells in the design or RAM cells with a block type setting of **MLAB** use LAB locations to implement memory.
Auto Register Duplication	Allows the Fitter to automatically duplicate registers within a LAB containing empty logic cells. This option does not alter the functionality of the design. The **Auto Register Duplication** option is also ignored if you select **OFF** as the setting for the **Logic Cell Insertion -- Logic Duplication** logic option. Turning on this option can allow the **Logic Cell Insertion -- Logic Duplication** logic option to improve a design's routability, but can make formal verification of a design more difficult.

Table 12-25: Advanced Fitter Settings (3 of 8)

Option	Description
Enable Bus Hold Circuitry	Enables bus-hold circuitry during device operation. If this option is turned on, a pin retains its last logic level when it is not driven, and does not go to a high impedance logic level. Do not use this option at the same time as **Weak Pull-Up Resistor** option. This option is ignored if it is applied to anything other than a pin.
Equivalent RAM and MLAB Paused Read Capabilities	Specifies whether RAMs implemented in MLAB cells must have equivalent pause read capabilities as RAMs implemented in block RAM. Pausing a read is the ability to keep around the last read value when reading is disabled. Allowing differences in paused read capabilities provides the Fitter more flexibility in implementing RAMs using MLAB cells. If this option is set to **Don't Care**, the Fitter may convert RAMs to MLAB cells even if they won't have equivalent paused read capabilities to a block RAM implementation. The Fitter also outputs an information message about RAMs with different paused read capabilities. If this option is set to **Care**, the Fitter does not convert RAMs to MLAB cells unless they have the equivalent paused read capabilities to a block RAM implementation. To allow the fitter the most flexibility in deciding which RAMs are implemented using MLAB cells, set this option to **Don't Care**.
Equivalent RAM and MLAB Power Up	Specifies whether RAMs implemented in MLAB cells must have equivalent power up conditions as RAMs implemented in block RAM. Power up conditions occur when the device is powered up or globally reset. Allowing non-equivalent power up conditions provides the Fitter more flexibility in implementing RAMs using MLAB cells. If this option is set to **Auto**, the Fitter may convert RAMs to MLAB cells even if they won't have equivalent power up conditions to a block RAM implementation. The Fitter also outputs a warning message about RAMs with non-equivalent power up conditions. If this option is set to **Don't Care**, the same behavior as **Auto** applies, but the message is an information message. If this option is set to **Care**, the Fitter does not convert RAMs to MLAB cells unless they have equivalent power up conditions to a block RAM implementation. To allow the fitter the most flexibility in deciding which RAMs are implemented using MLAB cells, set this option to **Auto** or **Don't Care**.
Final Placement Optimizations	Specifies whether the Fitter performs final placement optimizations. Performing final placement optimizations may improve timing and routability, but may also require longer compilation time. You can use the default setting of **Automatically** with the **Auto Fit** Fitter effort level (also the default) to allow the Fitter decide whether these optimizations should run based on the routability and timing requirements of the design.
Fitter Aggressive Routability Optimiza-tions	Specifies whether the Fitter aggressively optimizes for routability. Performing aggressive routability optimizations may decrease design speed, but may also reduce routing wire usage and routing time. The default **Automatically** setting allows the Fitter decide whether to perform these optimizations based on the routability and timing requirements of the design.

Table 12-26: Advanced Fitter Settings (4 of 8)

Option	Description
Fitter Effort	Specifies the level of physical synthesis optimization you want the Quartus Prime software to use when increasing the performance of a design, using the following options: • **Auto**—Auto Fit adjusts the fitter optimization effort to minimize compilation time, while still achieving the design timing requirements. Use the Auto Fit Effort Desired Slack Margin option to request that Auto Fit apply sufficient optimization effort to achieve additional timing margin. • **Standard**—Standard Fit uses maximum effort regardless of the design's requirements, leading to higher compilation time and more margin on easier designs. For difficult designs, Auto Fit and Standard Fit both use maximum effort. The Physical Synthesis Netlist Optimizations Report provides information about the physical synthesis optimizations performed.
Fitter Initial Placement Seed	Specifies the seed for the current design. The value can be any non-negative integer value. By default, the Fitter uses a seed of 1. The Fitter uses the seed as the initial placement configuration when attempting to optimize the design's timing requirements, including . Because each different seed value will result in a somewhat different fit, you can try several different seeds to attempt to obtain superior fitting results. The seeds that lead to the best fits for a design may change if the design changes. Also, changing the seed may or may not result in a better fit; therefore, you should specify a seed only if the Fitter is not meeting timing requirements by a small amount. **Note:** You can also use the Design Space Explorer II (DSEII) to sweep complex flow parameters, including the seed, in the Quartus Prime software to optimize design performance.
I/O Placement Optimizations	Specifies whether the Fitter optimizes the location of I/Os without pin assignments. Performing I/O placement optimizations may improve I/O timing, f_{MAX}, and fitting, but may also require longer compilation time.
Logic Cell Insertion	Allows the Fitter to automatically insert buffer logic cells between two nodes without altering the functionality of the design. Buffer logic cells are created from unused logic cells in the device. This option also allows the Fitter to duplicate a logic cell within a LAB when there are unused logic cells available in a LAB. Using this option can increase compilation time. The default setting of **Auto** allows these operations to run when 1) the design requires them to fit the design or 2) the performance of the design can be improved by this optimization with a nominal compilation time increase.

Option	Description
MLAB Add Timing Constraints for mixed-Port Feed-Through Mode Setting Don't Care	Specifies whether you want the TimeQuest Timing Analyzer to evaluate timing constraints between the write and the read operation of the MLAB memory block. Performing a write and read operation simultaneously at the same address might result in metastability because no timing constraints between those operations exist by default. Turning on this option introduces timing constraints between the write and read operation on the MLAB memory block and thereby avoids metastability issues; however, turning on this option degrades the performance of the MLAB memory blocks. If your design does not perform write and read operations simultaneously at the same address you do not need to set this option.

Table 12-27: Advanced Fitter Settings (5 of 8)

Option	Description
Optimize Design for Metastability	This setting improves the reliability of the design by increasing its Mean Time Between Failures (MTBF). When this setting is enabled, the Fitter increases the output setup slacks of synchronizer registers in the design, which can exponentially increase the design MTBF. This option only applies when using TimeQuest for timing-driven compilation. Use the TimeQuest `report_metastability` command to review the synchronizers detected in your design and to produce MTBF estimates.
Optimize Hold Timing	Directs the Fitter to optimize hold time within a device to meet timing requirements and assignments. The following settings are available: • **I/O Paths and Minimum TPD Paths**—Directs the Fitter to meet the following timing requirements and assignments: • • from I/O pins to registers. • from registers to I/O pins. • from I/O pins or registers to I/O pins or registers. • **All Paths**—Directs the Fitter to meet the following timing requirements and assignments: • • **tH** from I/O pins to registers. • Minimum t**CO** from registers to I/O pins. • Minimum t**PD** from I/O pins or registers to I/O pins or registers. This option is available for all Altera device families supported by the Quartus Prime software. When you turn off the **Optimize Timing** logic option, the **Optimize hold timing** option is not available.
Optimize IOC Register Placement for Timing	Specifies whether the Fitter optimizes I/O pin timing by automatically packing registers into I/Os to minimize I/O -> register and register -> I/O delays. When you select **Normal** , the Fitter opportunistically packs registers into I/Os that should improve I/O timing. When you enable **Pack All I/O Registers**, the Fitter aggressively packs any registers connected to input, output or output enable pins into I/Os, unless prevented by user constraints or other legality restrictions. By default, this option is set to **Normal**. This option requires enabling the **Optimize Timing** option.

Option	Description
Optimize Multi-Corner Timing	Directs the Fitter to consider all corner timing delays, including both fast-corner timing and slow-corner timing, during optimization to meet timing requirements at all process corners and operating conditions. By default, this option is **On**, and the Fitter optimizes designs considering multi-corner delays in addition to slow-corner delays, for example, from the fast-corner timing model, which is based on the fastest manufactured device, operating under high-voltage conditions. When this option is **Off**, the Fitter optimizes designs considering only slow-corner delays from the slow-corner timing model (slowest manufactured device for a given speed grade, operating in low-voltage conditions). Turning this option **On** typically helps to create a design implementation that is more robust across process, temperature, and voltage variations. This option is available for all Altera device families supported by the Quartus Prime software. When you turn **Off** the **Optimize Timing** option, the **Optimize multi-corner timing** option is not available.
Optimize Timing	Specifies whether the Fitter optimizes to meet the maximum delay timing requirements (for example, clock cycle time). By default, this option is set to **Normal compilation**. Turning it **Off** can help fit designs that have extremely high interconnect requirements and can also reduce compilation time, although at the expense of significant timing performance (since the Fitter ignores the design's timing requirements). If this option is **Off**, other Fitter timing optimization options have no effect (such as **Optimize Hold Timing**).

Table 12-28: Advanced Fitter Settings (6 of 8)

Option	Description
Optimize Timing for ECOs	Controls whether the Fitter optimizes to meet the user's maximum delay timing requirements (e.g.. clock cycle time, t_{SU}, t_{CO}) during ECO compiles. By default, this option is set to **Off**. Turning it on can improve timing performance at the cost of compilation time.
Perform Clocking Topology Analysis During Routing	Directs the Fitter to perform an analysis of the design's clocking topology and adjust the optimization approach on paths with significant clock skew. Enabling this option may improve hold timing at the cost of increased compile time.
Periphery to Core Placement and Routing Optimization	Specifies whether the Fitter should perform targeted placement and routing optimization on direct connections between periphery logic and registers in the FPGA core. If this option is set to **Auto**, the Fitter automatically identifies transfers with tight timing windows, places the core registers, and routes all connections to or from the periphery. The Compiler performs these placement and routing decisions before the rest of core placement and routing. This ensures that these timing-critical connections meet timing, and also avoids routing congestion. If this option is set to **On**, the Compiler optimizes all transfers between the periphery and core registers, regardless of timing requirements. Do not setting this option to **On** globally, rather use Assignment Editor to assign optimization to a targeted set of nodes or entities.

Option	Description
Placement Effort Multiplier	Specifies the relative time the Fitter spends in placement. The default value is 1.0 and legal values must be greater than 0. Specifying a floating-point number allows you to control the placement effort. A higher value increases CPU time but may improve placement quality. For example, a value of '4' increases fitting time by approximately 2 to 4 times but may increase quality.
PowerPlay Power Optimization During Fitting	Directs the Fitter to perform optimizations targeted at reducing the total power consumed by supported device families. The available settings for power-optimized fitting are: • **Off**—Performs no power optimizations. • **Normal compilation**—Performs power optimizations that are unlikely to adversely affect compilation time or design performance. • **Extra effort**—Performs additional power optimizations that might affect design performance or result in longer compilation time.

Table 12-29: Advanced Fitter Settings (7 of 8)

Option	Description
Programmable Power Maximum High-Speed Fraction of Used LAB Tiles	Sets the upper limit on the fraction of the high-speed LAB tiles. Legal values must be between 0.0 and 1.0. The default value is 1.0. A value of 1.0 means that there is no restriction on the number of high-speed tiles, and the Fitter uses the minimum number needed to meet the timing requirements of your design. Specifying a value lower than 1.0 might degrade timing quality, because some timing critical resources might be forced into low-power mode.
Programmable Power Technology Optimization	Controls how the Fitter configures tiles to operate in high-speed mode or low-power mode. **Automatic** specifies that the Fitter should minimize power without sacrificing timing performance. **Minimize Power Only** specifies that the Fitter should set the maximum number of tiles to operate in low-power mode. **Force All Used Tiles to High Speed** specifies that the Fitter sets all used tiles to operate in high-speed mode. **Force All Tiles with Failing Timing Paths to High Speed** specifies that the Fitter should ensure that all paths that are failing timing are set to high-speed mode. For designs that meet timing, the behavior of this setting should be similar to the **Automatic** setting. For designs that fail timing, all paths with negative slack are put in high-speed mode. Note that this likely does not increase the speed of the design, and it may increase static power consumption. This may assist in determining which logic paths need to be re-designed in order to close timing.
Regenerate Full Fit Reports During ECO Compiles	Controls whether the Fitter report is regenerated during ECO compiles. By default, this option is set to **Off**. Turning it **On** regenerates the report at the cost of compilation time.
Router Timing Optimization Level	Controls how aggressively the router tries to meet timing requirements. Setting this option to **Maximum** can increase design speed slightly, at the cost of increased compile time. Setting this option to **Minimum** can reduce compile time, at the cost of slightly reduced design speed. The default value is **Normal**.

Option	Description
Spectra-Q Physical Synthesis	Increases circuit performance by performing combinational and sequential optimization during fitting.
SSN Optimization	Specifies how aggressively the Fitter optimizes the design for simultaneous switching noise (SSN). If this option is set to **Off**, the Fitter does not perform any SSN optimizations. If this option is set to **Normal compilation**, the Fitter performs SSN optimizations which should not impact design performance. When this option is set to **Extra effort**, the Fitter performs aggressive SSN optimizations which may affect design performance.

Table 12-30: Advanced Fitter Settings (8 of 8)

Option	Description
Synchronizer Identification	Specifies how the TimeQuest Timing Analyzer identifies registers as being part of a synchronization register chain for metastability analysis. A synchronization register chain is a sequence of registers with the same clock with no fan-out in between, which is driven by a pin or logic from another clock domain. If this option is set to **Off**, the TimeQuest Timing Analyzer does not identify the specified registers, or the registers within the specified entity, as synchronization registers. If the option is set to **Auto**, the TimeQuest Timing Analyzer identifies valid synchronization registers that are part of a chain with more than one register that contains no combinational logic. If this option is set to **Forced if Asynchronous**, the TimeQuest Timing Analyzer identifies synchronization register chains if the software detects an asynchronous signal transfer, even if there is combinational logic or only one register in the chain. If this option is set to **Forced**, then the specified register, or all registers within the specified entity, are identified as synchronizers. Only apply the **Forced** option to the entire design. Otherwise, all registers in the design identify as synchronizers. Registers that are identified as synchronizers are optimized for improved Mean Time Between Failure (MTBF) as long as **Optimize Design for Metastability** is enabled. If a synchronization register chain is identified with the **Forced** or **Forced if Asynchronous** option, then the TimeQuest Timing Analyzer reports the metastability MTBF for the chain. MTBF is not reported for automatically-detected register chains; you can use the **Auto** setting to generate a report of possible synchronization chains in your design. If a synchronization register chain is identified with the **Forced** or **Forced if Asynchronous** option, then the TimeQuest Timing Analyzer reports the metastability MTBF for the chain when it meets the design timing requirements.
Treat Bidirectional Pin as Output Pin	Specifies that the bidirectional pin should be treated as an output pin, meaning that the input path feeds back from the output path.
Weak Pull-Up Resistor	Enables the weak pull-up resistor when the device is operating in user mode. This option pulls a high-impedance bus signal to VCC. Do not enable this option simultaneously with the **Enable Bus-Hold Circuitry** option. This option is ignored if it is applied to anything other than a pin.

Related Information

Running Place & Route on page 12-23

Document Revision History

Table 12-31: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• First version of document.

Related Information

Quartus Handbook Archive

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

Managing Metastability with the Quartus Prime Software 13

2015.11.02

QPP5V1 Subscribe Send Feedback

You can use the Quartus Prime 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 Prime 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 Prime 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 Prime 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

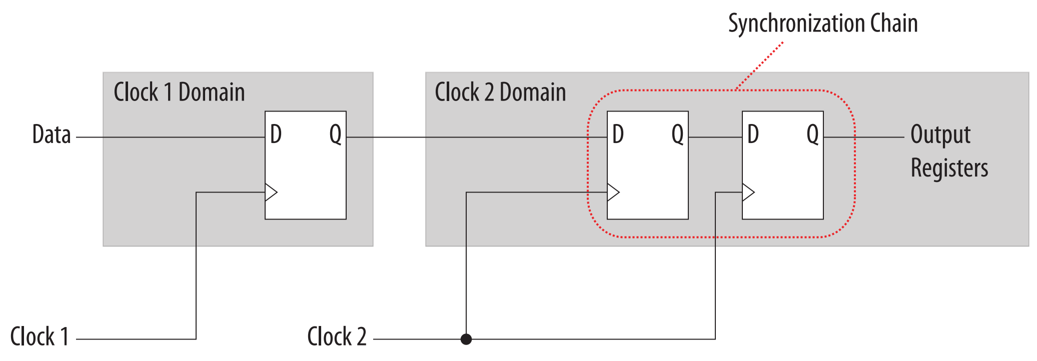
Synchronization Register Chains

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.

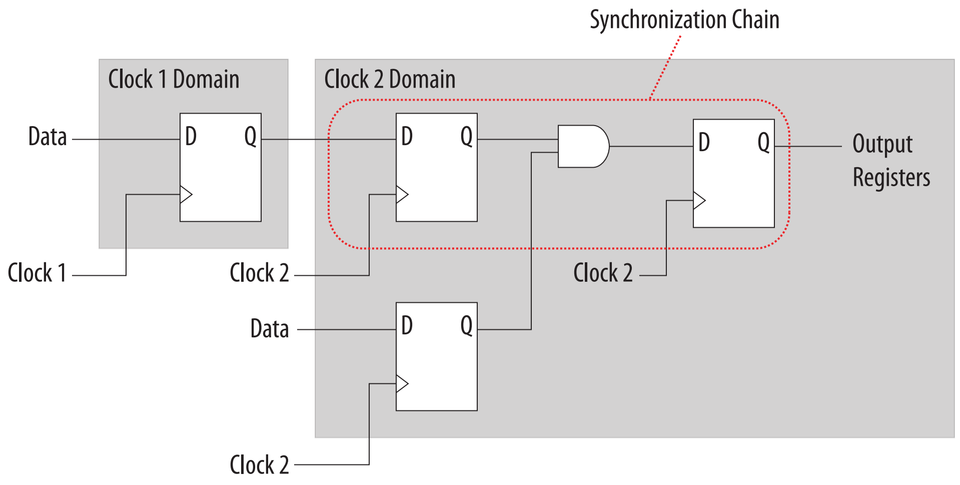
The length of the synchronization register chain is the number of registers in the synchronizing clock domain that meet the above requirements. The figure shows a sample two-register synchronization chain.

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 Prime 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 Prime 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 Prime 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

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 Prime 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 Prime 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 Prime 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 Prime 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 Prime Settings File Reference Manual
  For more information about settings and constraints in the Quartus Prime software

- **Command-Line Scripting**
  For more information about command-line scripting
- **About Quartus Prime 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 Prime 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 Prime software provides industry-leading analysis and optimization features to help you manage metastability in your FPGA designs. Set up your Quartus Prime 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 Prime software to make your design more robust with respect to metastability.

Document Revision History

Table 13-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.

Date	Version	Changes
June 2014	14.0.0	Updated formatting.
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 Handbook Archive
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Mitigating Single Event Upsets 14

2015.11.02

QPP5V1 Subscribe Send Feedback

The Quartus Prime 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 Prime 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

Some Altera devices offer dedicated error correcting code (ECC) circuitry for embedded memory blocks. The FIT rate for these memories can be reduced to near zero by enabling the ECC encode/decode blocks. On ingress, the ECC encoder adds 8 bits of redundancy to a 32 bit word. On egress, the 40 bit word is decoded back to 32 bits, and the redundant bits can be used to both detect and correct errors in the data resulting from SEU.

The existence of hard ECC and the strength of the ECC code (number of corrected and detected bits) varies by device family. Refer to the device handbook for details. If a device does not have a hard ECC block you can add ECC parity or use an ECC IP core.

For more information on embedded memories and ECC, refer to the *Embedded Memory (RAM: 1-PORT, RAM:2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*.

Related Information

- **Embedded Memory (RAM: 1-PORT, RAM:2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide**

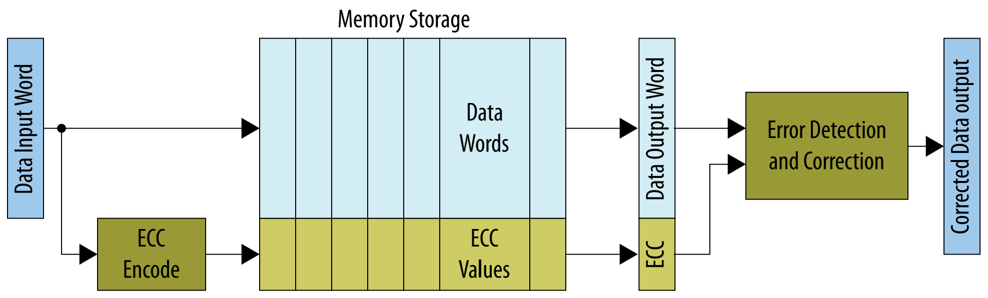
Configuring the ECCRAM

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 ECC 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 14-1: Memory Storage

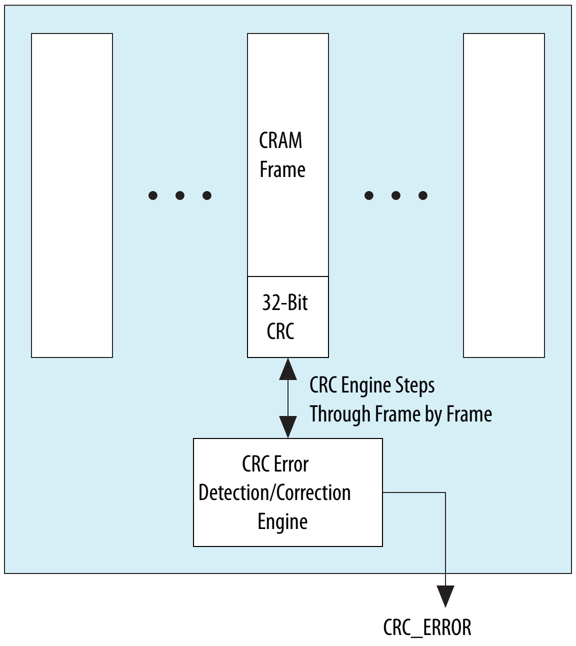


Mitigating SEU Effects in Configuration RAM

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 14-2: CRAM Frame


CRAM Frame
32-Bit CRC
CRC Engine Steps Through Frame by Frame
CRC Error Detection/Correction Engine
CRC_ERROR

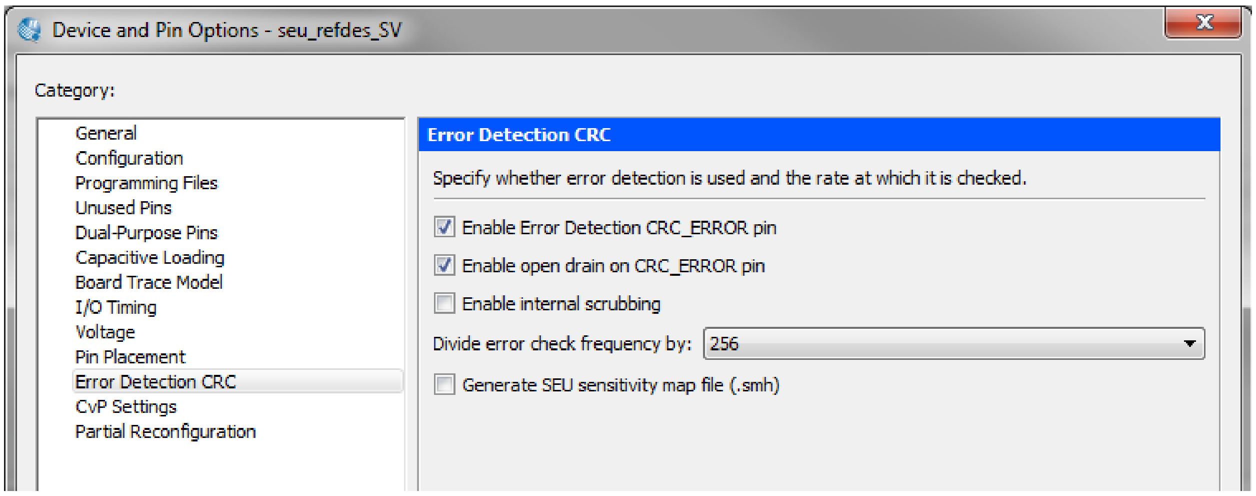
Related Information

Single Event Upsets

Scanning CRAM Frames

To enable the Quartus Prime 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 14-3: 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 Prime 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

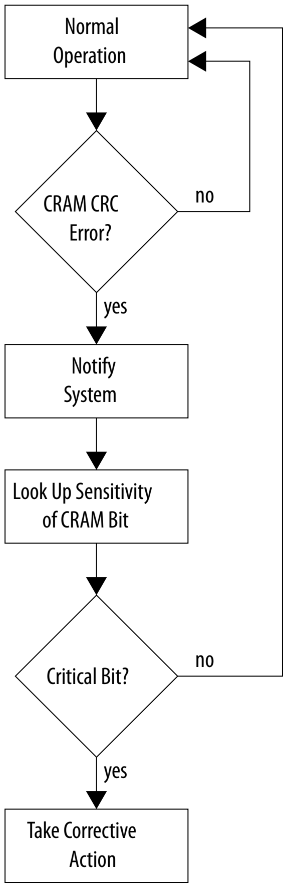
Understanding SEU Sensitivity

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 14-4: 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 Prime 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 Prime 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 Prime 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 14-1: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
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.

Optimizing the Design Netlist 15

2015.11.02

QPP5V1 Subscribe Send Feedback

This chapter describes how you can use the Quartus Prime 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 Prime RTL 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.

Related Information

- **When to Use the Netlist Viewers: Analyzing Design Problems** on page 15-1
- **Introduction to the User Interface** on page 15-5
- **Quartus Prime Design Flow with the Netlist Viewers** on page 15-2
- **RTL Viewer Overview** on page 15-3
- **Technology Map Viewer Overview** on page 15-4
- **Filtering in the Schematic View** on page 15-16
- **Cross-Probing to a Source Design File and Other Quartus Prime Windows** on page 15-21
- **Cross-Probing to the Netlist Viewers from Other Quartus Prime Windows** on page 15-21
- **Viewing a Timing Path** on page 15-22

When to Use the Netlist Viewers: Analyzing Design Problems

You can use the Netlist Viewers to analyze and debug your design. The following simple examples show how to use the RTL 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 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. 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.

Throughout your FPGA design, debug, and optimization stages, you can use all of the netlist viewers in many ways to increase your productivity while analyzing a design.

Related Information

- **Quartus Prime Design Flow with the Netlist Viewers** on page 15-2
- **RTL Viewer Overview** on page 15-3
- **Technology Map Viewer Overview** on page 15-4

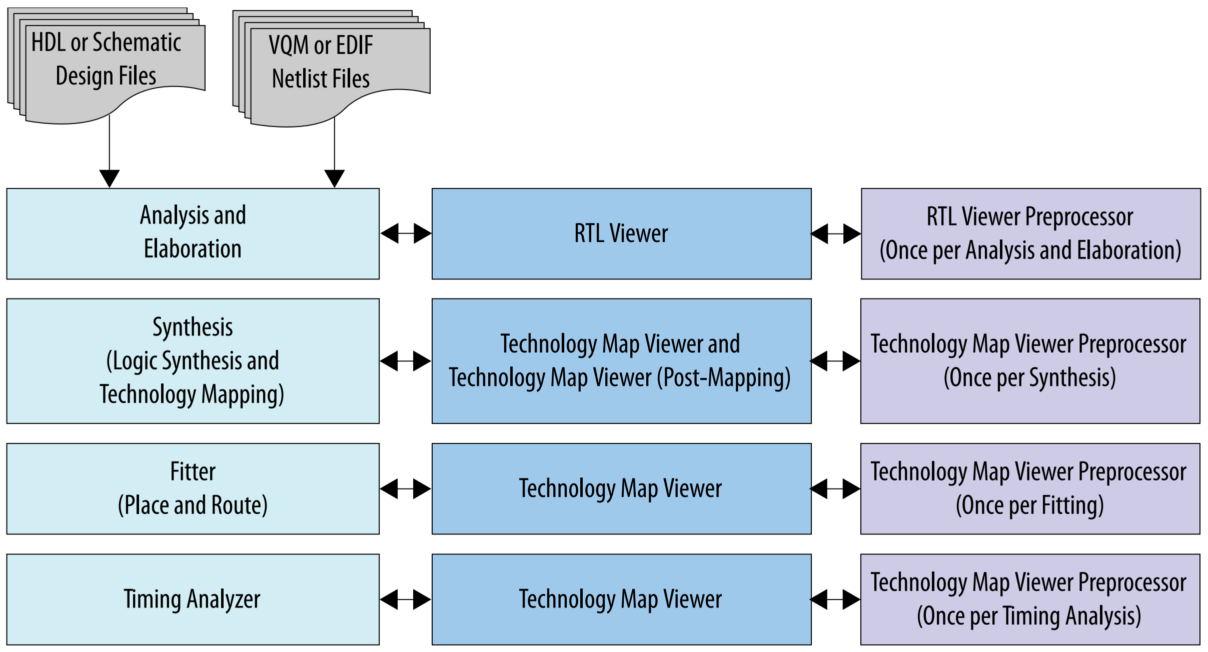
Quartus Prime Design Flow with the Netlist Viewers

When you first open one of the Netlist Viewers after compiling the design, a preprocessor stage runs automatically before the Netlist Viewer opens.

Click the link in the preprocessor process box to go to the **Settings** > **Compilation Process Settings** page where you can turn on the **Run Netlist Viewers preprocessing during compilation** option. If you turn this option on, the preprocessing becomes part of the full project compilation flow and the Netlist Viewer opens immediately without displaying the preprocessing dialog.

Figure 15-1: Quartus Prime Design Flow Including the RTL Viewer and Technology Map Viewer

This figure shows how Netlist Viewers fit into the basic Quartus Prime 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)
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 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 Prime 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 RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your Quartus Prime integrated synthesis results or your third-party netlist file in the Quartus Prime software.

You can view results after Analysis and Elaboration when your design uses any supported Quartus Prime 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 RTL Viewer displays a schematic view of the design netlist after Analysis and Elaboration or netlist extraction is performed by the Quartus Prime software, but before technology mapping and any synthesis or fitter optimizations. This view a preliminary pre-optimization design structure and closely represents your original source design.

- If you synthesized your design with the Quartus Prime integrated synthesis, this view shows how the Quartus Prime software interpreted your design files.
- If you use a third-party synthesis tool, this view shows the netlist written by your synthesis tool.

While displaying your design, the RTL Viewer optimizes the netlist to maximize readability:

- Removes logic with no fan-out (unconnected output) or fan-in (unconnected inputs) from the display.
- Hides default connections such as V_{CC} and GND.
- Groups pins, nets, wires, module ports, and certain logic into buses where appropriate.
- Groups constant bus connections are grouped.
- Displays values in hexadecimal format.
- Converts NOT gates into bubble inversion symbols in the schematic.
- Merges chains of equivalent combinational gatesinto 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.

To run the RTL Viewer for a Quartus Prime project, first analyze the design to generate an RTL netlist. To analyze the design and generate an RTL netlist, from the Processing menu, click **Start** > **Start Analysis & Elaboration**. You can also perform a full compilation on any process that includes the initial Analysis and Elaboration stage of the Quartus Prime compilation flow.

To open the RTL Viewer, from the Tools menu click**Netlist Viewers** > **RTL Viewer**.

Related Information

Introduction to the User Interface on page 15-5

Technology Map Viewer Overview

The Quartus Prime 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 device 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 Quartus Prime software maintains the port names of each hierarchy throughout synthesis. However, the software may change or remove port names from the design. For example, if a port is unconnected or driven by GND or V_{CC}, the software removes it during synthesis. If a port name changes, the software assigns a related user logic name in the design or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus Prime technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus Prime 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 15-16
- **Viewing a Timing Path** on page 15-22
- **Introduction to the User Interface** on page 15-5

Schematic Viewer

The Quartus Prime Schematic 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 Schematic Viewer shows the hierarchy of atom primitives (such as device logic cells and I/O ports) in your design. 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 Quartus Prime software maintains the port names of each hierarchy throughout synthesis. However, the software may change or remove port names from the design. For example, if a port is unconnected or driven by GND or V_{CC}, the software removes it during synthesis. If a port name changes, the software assigns a related user logic name in the design or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus Prime schematic after synthesis, fitting, or timing analysis.

To open the Schematic Viewer, point to **Tools** > **Netlist Viewers** > **Select Snapshot...**. The **Select Snapshot...** dialog box appears where you can select from a **synthesized**, **planned**, **placed**, **routed**, or **final** view.

Controls and options are the same as those used by the RTL Viewer and Technology Map Viewer.

Related Information

Introduction to the User Interface on page 15-5

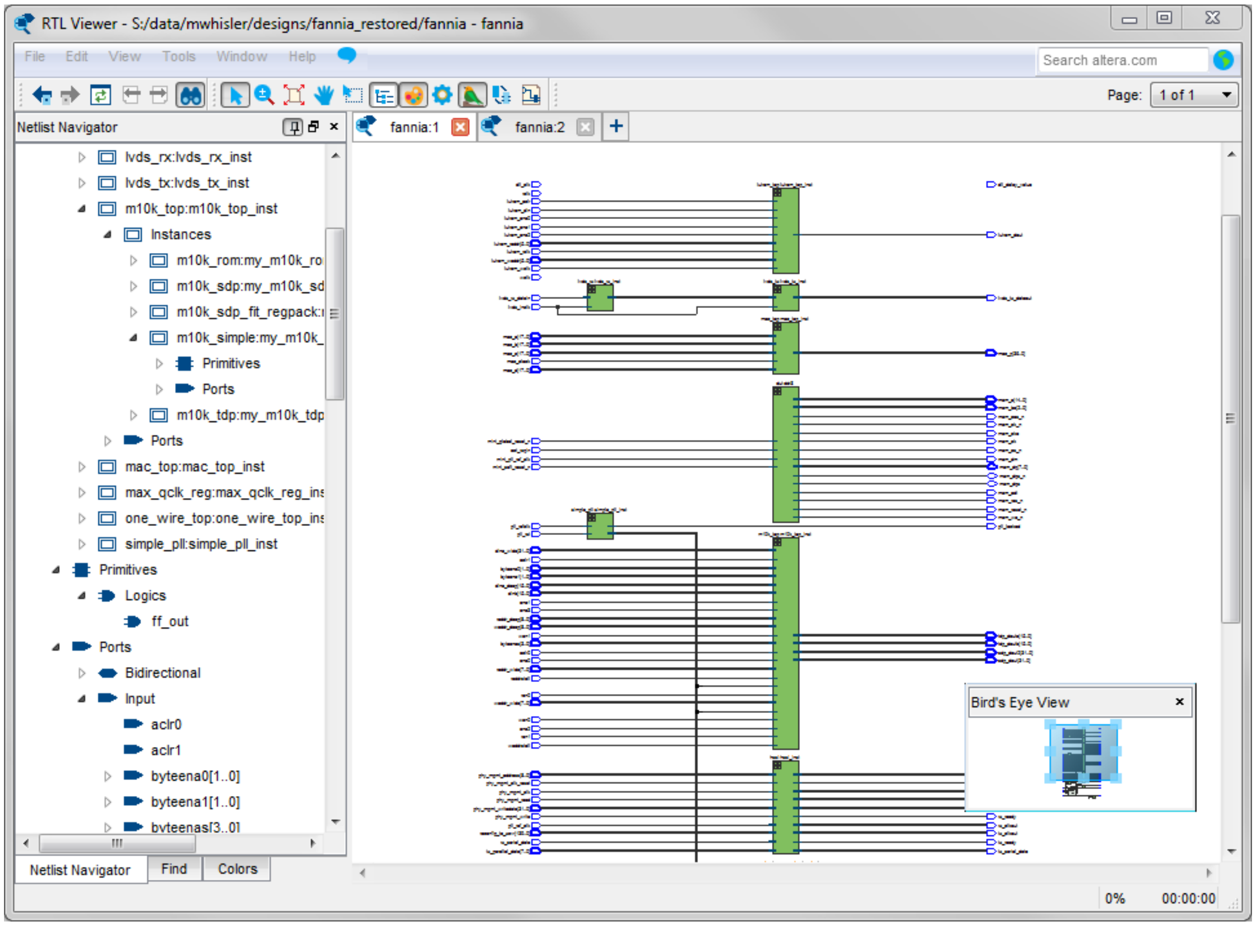
Introduction to the User Interface

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 15-2: RTL Viewer

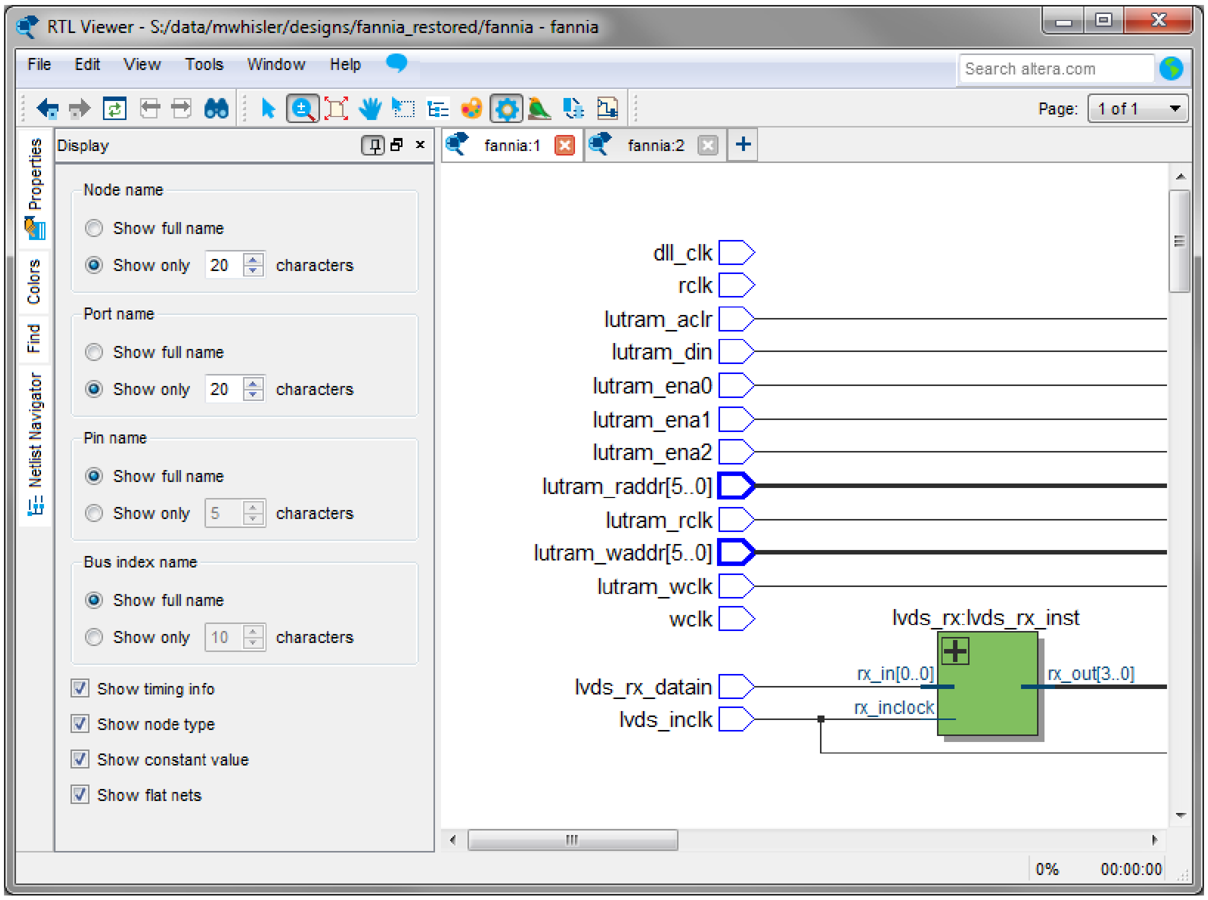


Netlist Viewers also contain a toolbar that provides tools to use in the schematic view.

- 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 zoom actions.
- The **Refresh** button to restore the schematic view and optimizes the layout. **Refresh** does not reload the database if you change your design and recompile.
- Click the **Find** button opens and closes the **Find** pane.
- Click the **Selection Tool** and **Zoom Tool** buttons to toggle between the selection mode and zoom mode.

- Click the **Fit in Page** button resets the schematic view to encompass the entire design.
- Use the **Hand Tool** to change the focus of the veiwer without changing the perspective.
- Click the **Area Selection Tool** to drag a selection box around ports, pins, and nodes in an area.
- Click the **Netlist Navigator** button to open or close the **Netlist Navigator** pane.
- Click the **Color Settings** button to open the **Colors** pane where you can customize the Netlist Viewer color scheme.
- Click the **Display Settings** button to open the **Display** pane where you can specify the following settings:
    - **Show full name** or **Show only <n> characters**. You can specify this separately for **Node name**, **Port name**, **Pin name**, or **Bus name**.
    - Turn **Show timing info** on or off.
    - Turn **Show node type** on or off.
    - Turn **Show constant value** on or off.
    - Turn **Show flat nets** on or off.

Figure 15-3: Display Settings



RTL Viewer - S:/data/mwhisler/designs/fannia_restored/fannia - fannia

Display

Node name
- ( ) Show full name
- (●) Show only 20 characters

Port name
- ( ) Show full name
- (●) Show only 20 characters

Pin name
- (●) Show full name
- ( ) Show only 5 characters

Bus index name
- (●) Show full name
- ( ) Show only 10 characters

- ☑ Show timing info
- ☑ Show node type
- ☑ Show constant value
- ☑ Show flat nets

- 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. Instance pins are hidden by default.
- 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), ande one Technology Map Viewer (Post-Mapping) 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

- **RTL Viewer Overview** on page 15-3
- **Technology Map Viewer Overview** on page 15-4
- **Schematic Viewer** on page 15-5
- **Netlist Navigator Pane** on page 15-8
- **Netlist Viewers Find Pane** on page 15-11
- **Properties Pane** on page 15-9

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 15-1: Netlist Navigator Pane Elements

Elements	Description
Instances	Modules or instances in the design that can be expanded to lower hierarchy levels.

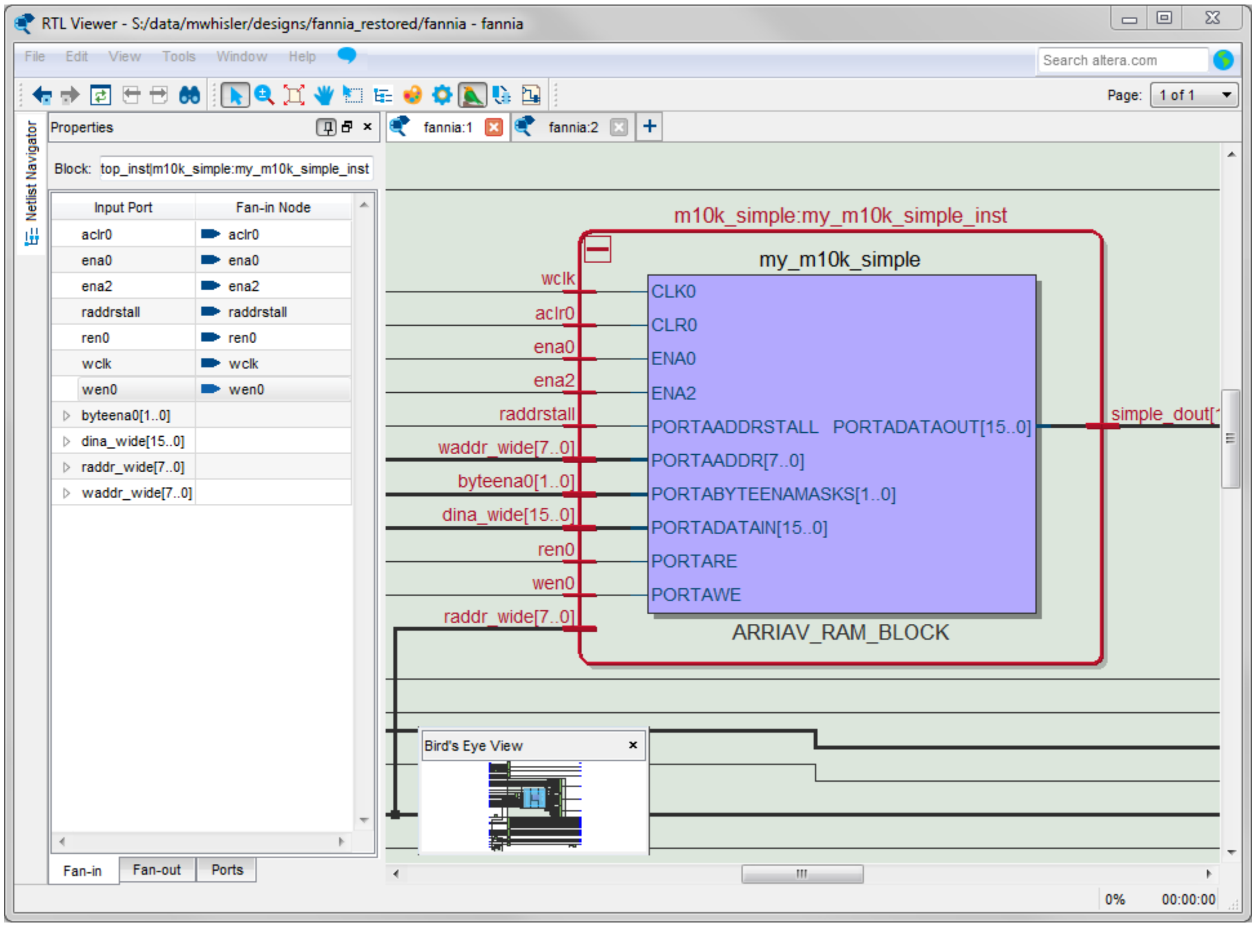
Elements	Description
Primitives	Low-level nodes that cannot be expanded to any lower hierarchy level. These primitives include: • Registers and gates that you can view in the RTL Viewer when using Quartus Prime integrated synthesis • Logic cell atoms in the Technology Map Viewer or in the RTL Viewer 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 15-4: 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**.



Block: top_inst|m10k_simple:my_m10k_simple_inst

Input Port	Fan-in Node
aclr0	aclr0
ena0	ena0
ena2	ena2
raddrstall	raddrstall
ren0	ren0
wclk	wclk
wen0	wen0
byteena0[1..0]	
dina_wide[15..0]	
raddr_wide[7..0]	
waddr_wide[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 15-2: 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)
--	The port is connected and has value (other than V_{CC} or GND)

Value	Description
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.

Right-click in a tab to see a shortcut menu to perform the following actions:

- 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

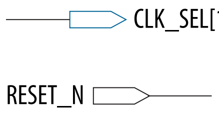
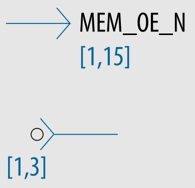
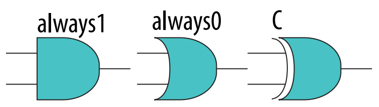
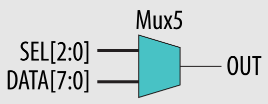
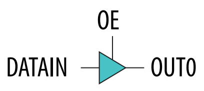
Schematic Symbols

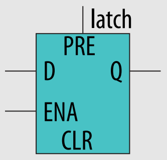
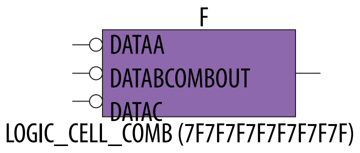
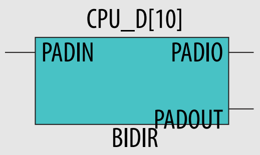
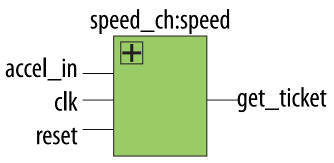
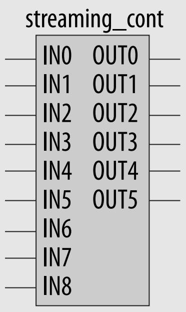
The symbols for nodes in the schematic represent elements of your design netlist. These elements include input and output ports, registers, logic gates, Altera primitives, high-level operators, and hierarchical instances.

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 15-3: Symbols in the Schematic View

This 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.

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.
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.

Symbol	Description
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.
Ecrypted Instance  streaming_cont IN0 OUT0 IN1 OUT1 IN2 OUT2 IN3 OUT3 IN4 OUT4 IN5 OUT5 IN6 IN7 IN8	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.

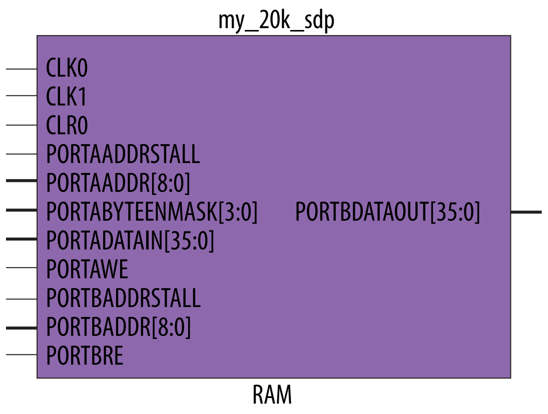
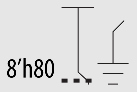
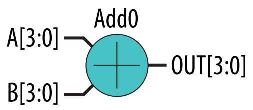
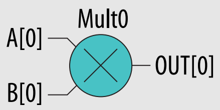
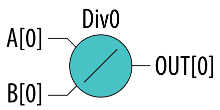
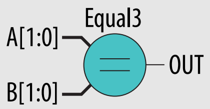
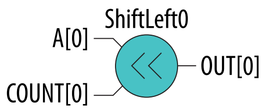
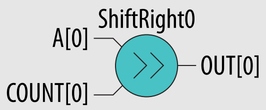
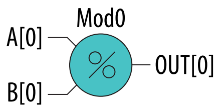
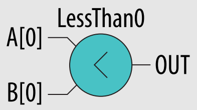
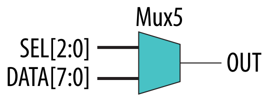
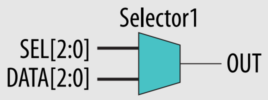
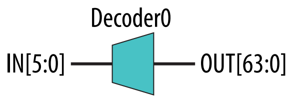
Symbol	Description
RAM  my_20k_sdp CLK0 CLK1 CLR0 PORTAADDRSTALL PORTAADDR[8:0] PORTABYTEENMASK[3:0] PORTADATAIN[35:0] PORTAWE PORTBADDRSTALL PORTBADDR[8:0] PORTBRE PORTBDATAOUT[35:0] RAM	A synchronous memory instance with registered inputs and optionally registered outputs. The symbol shows the device 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.

Table 15-4: Operator Symbols in the RTL Viewer Schematic View

The following lists and describes the additional higher level 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)`

Symbol	Description
	A modulo operator: `OUT = (A%B)`
	A less than comparator: `OUT = (A<:B:A>B)`
	A multiplexer: `OUT = DATA [SEL]` The data range size is $2^{\text{sel range size}}$
	A selector: A multiplexer with one-hot select input and more than two input signals
	A binary number decoder: `OUT` = (binary_number (`IN`) == x) for x = 0 to x = $2^{(n+1)}$ - 1

Related Information

- **Partition the Schematic into Pages** on page 15-20
- **Follow Nets Across Schematic Pages** on page 15-20

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 15-8

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 command 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.

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, or ports of a node, 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.

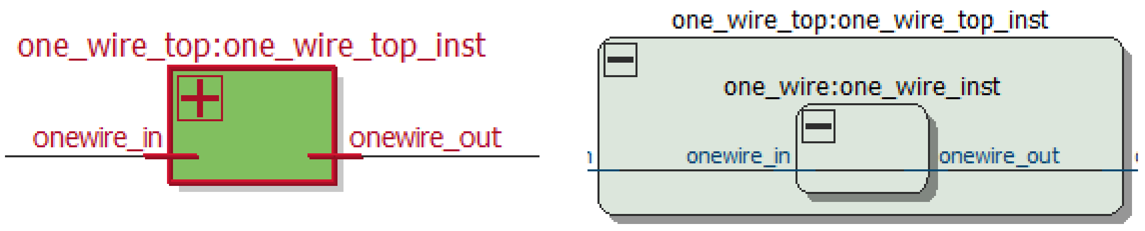
View Contents of Nodes in the Schematic View

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 15-5: 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 15-6: 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 .

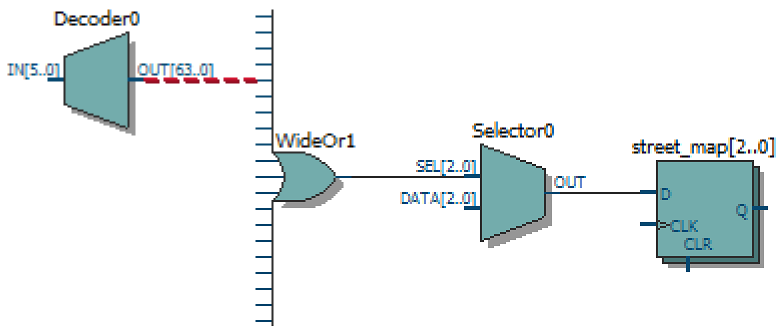


Figure 15-7: Display Nets Across Hierarchies

In cases where the net connects to an instance outside the hierarchy, you can select the net, and unwrap the node to see the destination ports.

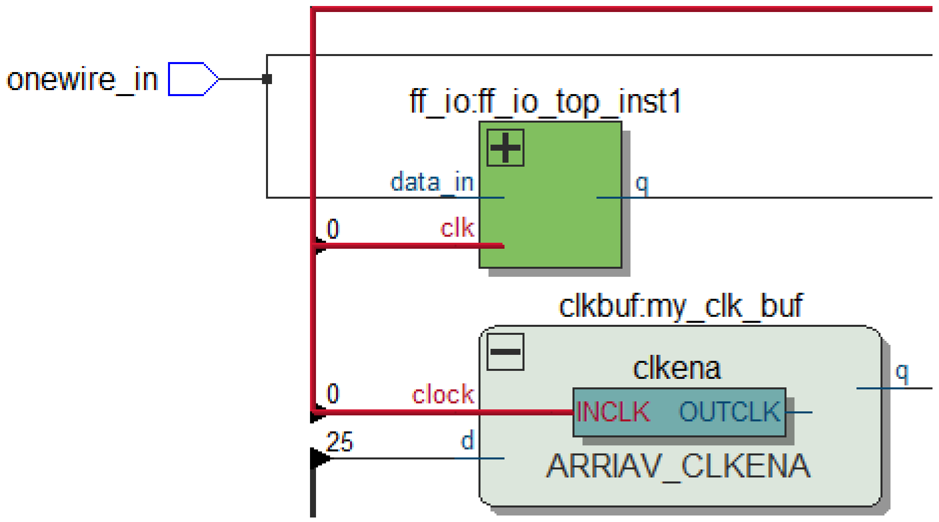
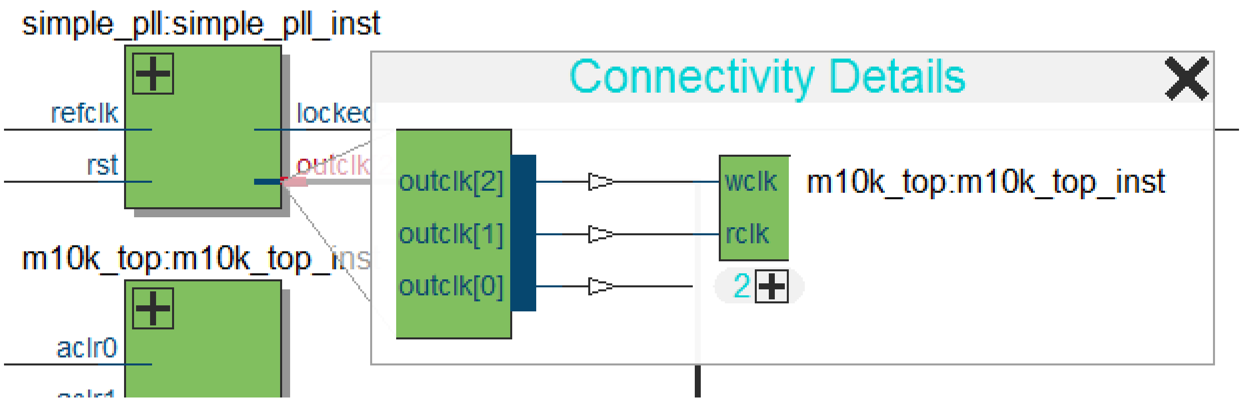


Figure 15-8: Show Connectivity Details

You can select a bus port or bus pin and click **Connectivity Details** in the context menu for that object.



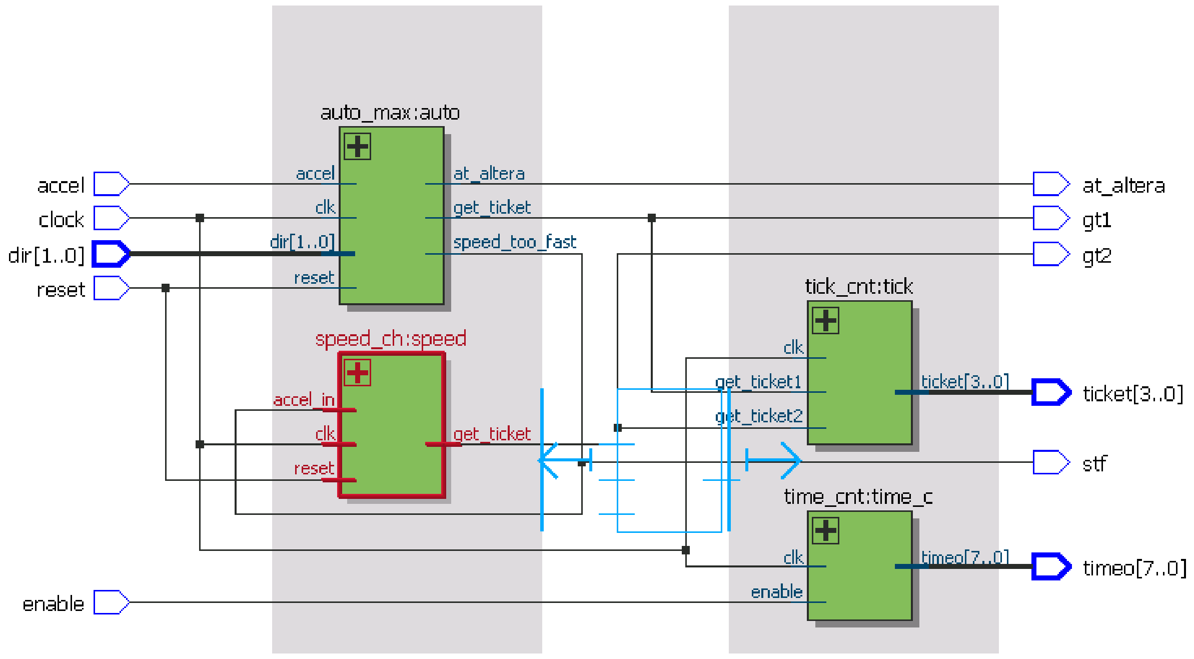
You can double-click on objects in the **Connectivity Details** window to navigate to them quickly. If the plus symbol appears, you can further unwrap objects in the view. This can be very useful when tracing a signal in a complex netlist.

Moving Nodes in the Schematic View

You can drag and drop items in the schematic view to rearrange them.

Figure 15-9: 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 15-9

Zoom Controls

Use the Zoom Tool in the toolbar, or mouse gestures, to control the magnification of your schematic on the View menu.

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).

You can 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.

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.

Related Information

Filtering in the Schematic View on page 15-16

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 Prime 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 15-5

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 navigate across pages.

Related Information

Schematic Symbols on page 15-12

Cross-Probing to a Source Design File and Other Quartus Prime Windows

The RTL Viewer and Technology Map Viewer allow you to cross-probe to the source design file and to various other windows in the Quartus Prime 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 Prime 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.

Cross-Probing to the Netlist Viewers from Other Quartus Prime Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows in the Quartus Prime 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 and Technology Map Viewer, and you can locate nodes in the RTL Viewer and Technology Map Viewer from the following Quartus Prime 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 Prime 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.

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, then click **Filter** > **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 click**Custom Reports** > **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
2015.11.02	15.1.0	Added Schematic Viewer topic for viewing stage snapshots. Added information for the following new features and feature updatess: • Nets visible across hierarchies • Connection Details • Display Settings • Hand Tool • Area Selection Tool • New default behavior for Show/Hide Instance Pins (default is now off)
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 Prime software version 10.0, including major user interface changes
November 2009	9.1.0	• Updated devices • Minor text edits
March 2009	9.0.0	• Chapter 13 was formerly Chapter 12 in version 8.1.0 • Updated Figure 13–2, Figure 13–3, Figure 13–4, Figure 13–14, and Figure 13–30 • Added “Enable or Disable the Auto Hierarchy List” on page 13–15 • Updated “Find Command” on page 13–44

Date	Version	Changes
November 2008	8.1.0	Changed page size to 8.5” × 11”
May 2008	8.0.0	• Added Arria GX support • Updated operator symbols • Updated information about the radial menu feature • Updated zooming feature • Updated information about probing from schematic to SignalTap II Analyzer • Updated constant signal information • Added .png and .gif to the list of supported image file formats • Updated several figures and tables • Added new sections “Enabling and Disabling the Radial Menu”, “Changing the Time Interval”, “Changing the Constant Signal Value Formatting”, “Logic Clouds in the RTL Viewer”, “Logic Clouds in the Technology Map Viewer”, “Manually Group and Ungroup Logic Clouds”, “Customizing the Shortcut Commands” • Renamed several sections • Removed section “Customizing the Radial Menu” • Moved section “Grouping Combinational Logic into Logic Clouds” • Updated document content based on the Quartus Prime software version 8.0

Related Information

Quartus Handbook Archive

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

Mentor Graphics Precision Synthesis Support 16

2015.11.02

QPP5V1 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 Prime 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 Prime software.

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 Prime 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 Prime project and import the following files generated by the Precision Synthesis software into the Quartus Prime project:

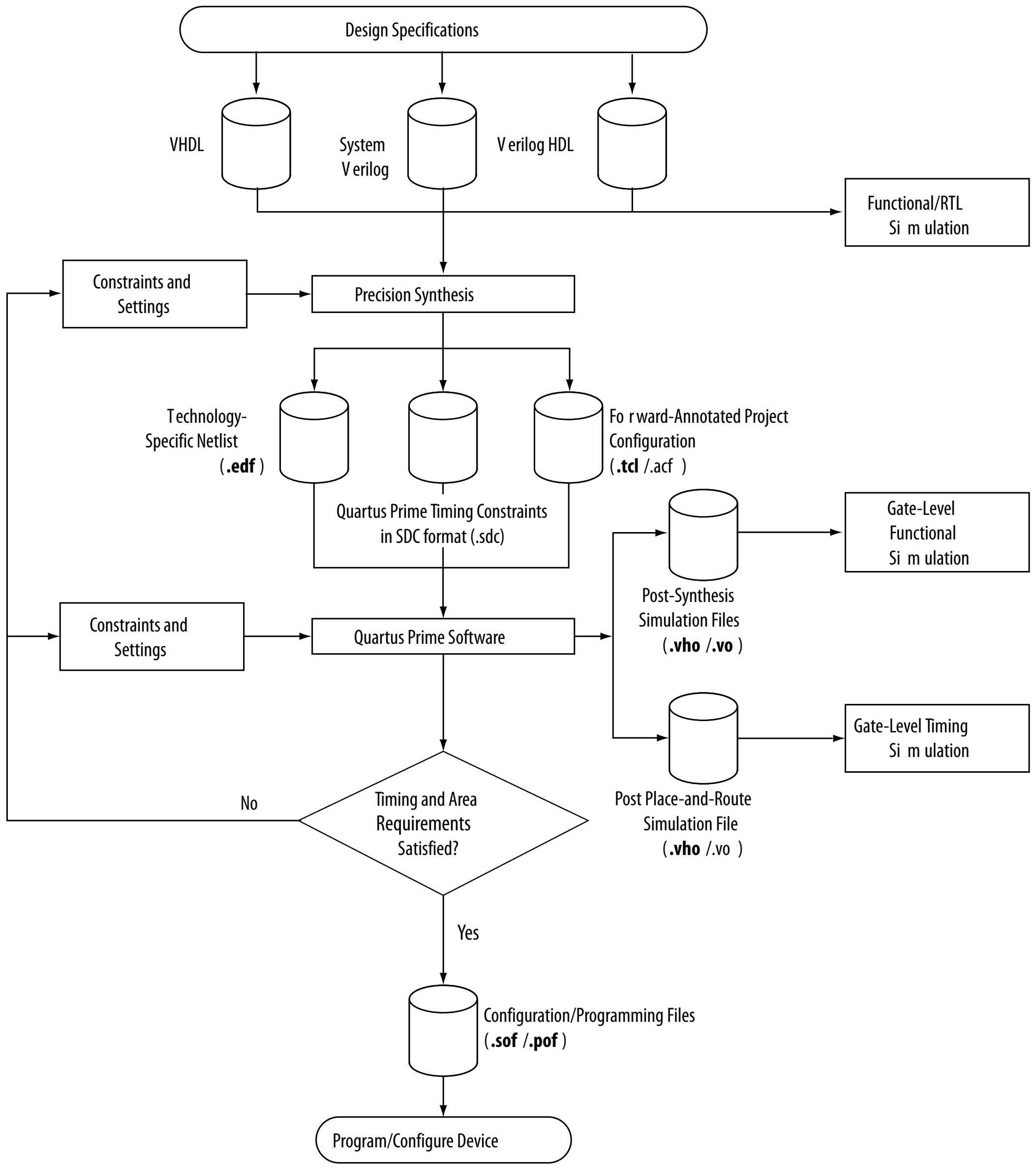
- The Verilog Quartus Mapping File (**.vqm**) netlist
- Synopsys Design Constraints File (**.sdc**) for TimeQuest Timing Analyzer constraints
- Tcl Script Files (**.tcl**) to set up your Quartus Prime project and pass constraints

Note: If your design uses the Classic Timing Analyzer for timing analysis in the Quartus Prime 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 Prime 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 Prime software from within the Precision Synthesis software, or run the Precision Synthesis software using the Quartus Prime software.

Figure 16-1: Design Flow Using the Precision Synthesis Software and Quartus Prime 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 Prime Timing Constraints in SDC format (.sdc)
Post-Synthesis Simulation Files (.vho/.vo)
Gate-Level Functional Simulation
Constraints and Settings
Quartus Prime 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 Prime Software from within the Precision Synthesis Software on page 16-9
- Using the Quartus Prime Software to Run the Precision Synthesis Software on page 16-10

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 Prime software. Repeat the process until the area and timing requirements are met.

You can use other options and techniques in the Quartus Prime software to meet area and timing requirements. For example, the **WYSIWYG Primitive Resynthesis** option can perform optimizations on your EDIF netlist in the Quartus Prime 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 Prime 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 16-1: Precision Synthesis Software Intermediate and Output Files

File Extension	File Description
.psp	Precision Synthesis Project File.
.xdb	Mentor Graphics Design Database File.
.rep(10)	Synthesis Area and Timing Report File.
.vqm(11)	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.

(10) The timing report file includes performance estimates that are based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus Prime 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 Prime software after place-and-route for final resource utilization results.

(11) The Precision Synthesis software-generated VQM file is supported by the Quartus Prime 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 Prime Project Configuration file and is used to make basic project and placement assignments, and to create and compile a Quartus Prime 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 Prime 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 Prime software, and has the naming convention <*project name*>**_pnr_constraints .sdc**.

Related Information

- **Exporting Designs to the Quartus Prime Software Using NativeLink Integration** on page 16-9
- **Synthesizing the Design and Evaluating the Results** on page 16-8

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 Prime software, placement constraints are written in a **.tcl** file and timing constraints for the TimeQuest Timing Analyzer are written in the Quartus Prime **.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 Prime 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 Prime 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 16-2: 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 16-1: Verilog HDL Pin Assignment

```
//pragma attribute clk pin_number P10;
```

Example 16-2: 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 Prime 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 Prime 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 Prime 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 Prime Software Using NativeLink Integration

The NativeLink feature in the Quartus Prime software facilitates the seamless transfer of information between the Quartus Prime software and EDA tools, which allows you to run other EDA design entry/ synthesis, simulation, and timing analysis tools automatically from within the Quartus Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime executables, such as analysis and synthesis, Fitter, and the TimeQuest Timing Analyzer for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus Prime Modular** flow option in the Precision Synthesis software. By default, the Precision Synthesis software generates a Quartus Prime Project Configuration

File (**.tcl** file) for current device families. Timing constraints that you set during synthesis are exported to the Quartus Prime place-and-route constraints file <*project name*>**_pnr_constraints.sdc**.

After you compile the design in the Quartus Prime software from within the Precision Synthesis software, you can invoke the Quartus Prime GUI manually and then open the project using the generated Quartus Prime project file. You can view reports, run analysis tools, specify options, and run the various processing flows available in the Quartus Prime software.

For more information about running the Quartus Prime software from within the Precision Synthesis software, refer to the *Altera Quartus Prime Integration* chapter in the *Precision Synthesis Reference Manual.*

Running the Quartus Prime Software Manually Using the Precision Synthesis-Generated Tcl Script

You can run the Quartus Prime 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 Prime 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 Prime software.

Using the Quartus Prime Software to Run the Precision Synthesis Software

With NativeLink integration, you can set up the Quartus Prime 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 that you have set in the Quartus Prime software.

Related Information

- **Exporting Designs to the Quartus Prime Software Using NativeLink Integration** on page 16-9
- **Using the NativeLink Feature with Other EDA Tools online help**

Passing Constraints to the Quartus Prime 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 Prime 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 16-3: 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 Prime software.

The Quartus Prime 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 16-4: 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 Prime 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 Prime software, and a message is displayed.

set_output_delay

This port-specific output delay constraint is specified in the Precision Synthesis software.

Example 16-5: 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 Prime 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 Prime 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 16-6: Using the set_max_delay Constraint

```
set_max_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

Example 16-7: 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 16-8: 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 Prime 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 Prime software with a `through` path specification.

set_multicycle_path

The multicycle path constraint is specified in the Precision Synthesis software.

Example 16-9: 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 Prime 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 Prime software automatically expands all wildcards.

Any multicycle path setting in Precision Synthesis software can be mapped to a setting in the Quartus Prime 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 Prime 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 16-16
- **Recommended HDL Coding Styles documentation** on page 11-1
- **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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 16-10: 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 16-11: 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 11-1

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 16-3: 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 16-12: Setting the dedicated_mult Attribute in Verilog HDL

```
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 16-13: 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 16-14: Setting the preserve_signal Attribute in Verilog HDL

```
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 16-15: Setting the preserve_signal Attribute in VHDL

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 16-16: 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 16-17: 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 11-1

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 Prime 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 16-4: 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 16-18: Setting the extract_mac Attribute in Verilog HDL

```
//synthesis attribute <module name> extract_mac <value>
```

Example 16-19: 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 Prime software.

Example 16-20: 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 16-21: 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 11-1

Document Revision History

Table 16-5: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
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 Prime 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 Prime software version 10.0 release
November 2009	9.1.0	• Minor updates for the Quartus Prime software version 9.1 release
March 2009	9.0.0	• Updated list of supported devices for the Quartus Prime 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 Prime software version 8.0

Related Information

Quartus Handbook Archive

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

Synopsys Synplify Support 17

2014.11.02

QPP5V1 Subscribe Send Feedback

About Synplify Support

This manual delineates the support for the Synopsys Synplify software in the Quartus Prime 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 Prime software
- Exporting designs and constraints to the Quartus Prime 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

Related Information

- **Synplify Synthesis Techniques with the Quartus Prime Software online training**
- **Synplify Pro Tips and Tricks online training**

Design Flow

The following steps describe a basic Quartus Prime software design flow using the Synplify software:

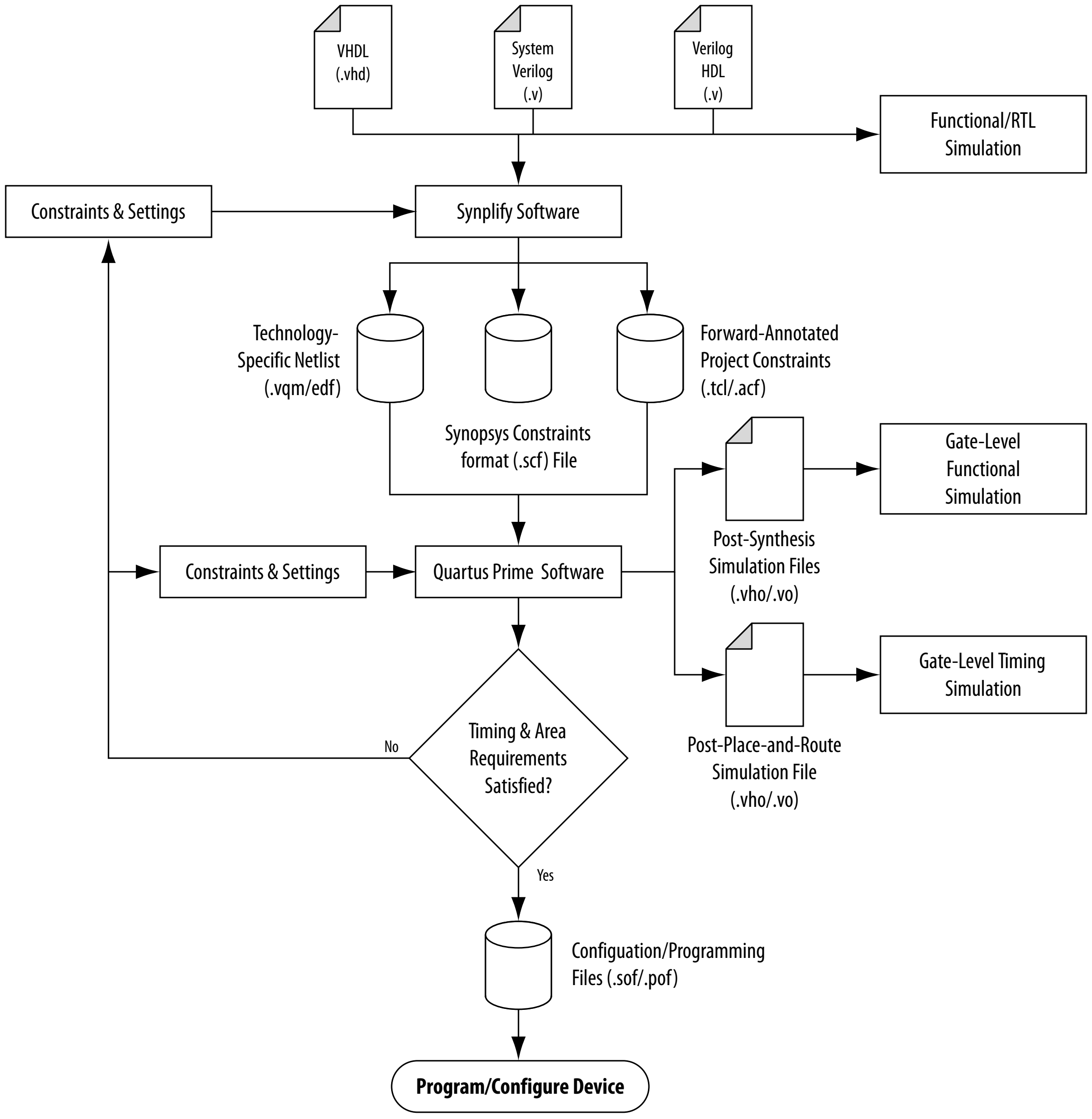
1. Create Verilog HDL or VHDL design files.
2. Set up a project in the Synplify software and add the HDL design files for synthesis.
3. Select a target device and add timing constraints and compiler directives in the Synplify software to help optimize the design during synthesis.
4. Synthesize the project in the Synplify software.
5. Create a Quartus Prime project and import the following files generated by the Synplify software into the Quartus Prime 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 Prime project and pass constraints.

Note: Alternatively, you can run the Quartus Prime 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 Prime Software
Gate-Level Timing Simulation
Post-Place-and-Route Simulation File (.vho/.vo)
Timing & Area Requirements Satisfied?
No
Yes
Configuation/Programming Files (.sof/.pof)
Program/Configure Device

Related Information

- **Running the Quartus Prime 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 Prime 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 Prime Software Version

You can specify your version of the Quartus Prime 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 Prime software whenever possible. If your Quartus Prime 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 Prime 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 Prime Software Version at the Command Line

```
set_option -quartus_version <version number>
```

Exporting Designs to the Quartus Prime Software Using NativeLink Integration

The NativeLink feature in the Quartus Prime software facilitates the seamless transfer of information between the Quartus Prime 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 Prime 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 Prime software for place-and-route. You can run the Quartus Prime software from within the Synplify software or as a stand-alone application. After you import the design into the Quartus Prime 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 Prime 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 Prime software with a single GUI for both synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus Prime software from within the Synplify software GUI, or to run the Synplify software from within the Quartus Prime software GUI.

Running the Quartus Prime Software from within the Synplify Software

To run the Quartus Prime software from within the Synplify software, you must set the *QUARTUS_ROOTDIR* environment variable to the Quartus Prime software installation directory located in <*Quartus Prime system directory*>\altera\ <*version number*>**\quartus**. You must set this environment variable to use the Synplify and Quartus Prime software together. Synplify also uses this variable to open the Quartus Prime 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 Prime 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 Prime menu to run the Quartus Prime software at any time following a successful completion of synthesis. In the Synplify software, on the Options menu, click **Quartus Prime** and then choose one of the following commands:

- **Launch Quartus** —Opens the Quartus Prime software GUI and creates a Quartus Prime 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 Prime commands.
- **Run Background Compile**—Runs the Quartus Prime 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 Prime project and directs the <*project_name*>**.tcl** file to pass constraints from the Synplify software to the Quartus Prime 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 Prime Software to Run the Synplify Software

You can set up the Quartus Prime 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 Prime software. When you use this feature, the Synplify software does not use any timing constraints or assignments that you have set in the Quartus Prime 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 Prime software, instead of opening the Synplify software from within the Quartus Prime software.

To set up the Quartus Prime 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 Prime 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 Prime software.
.scf(12)	Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer.

(12) If your design uses the Classic Timing Analyzer for timing analysis in the Quartus Prime software versions 10.0 and earlier, the Synplify software generates timing constraints in the Tcl Constraints File (**.tcl**). If you are using the Quartus Prime 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 Prime software is created for all devices. The **.tcl** file contains the appropriate Tcl commands to create and set up a Quartus Prime 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(13)	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 Prime software.

After synthesis is complete, do the following steps:

1. Import the **.vqm** netlist to the Quartus Prime 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

(13) This report file includes performance estimates that are often based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus Prime 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 Prime 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 Prime software after place-and-route.

Running the Quartus Prime Software Manually With the Synplify-Generated Tcl Script

You can run the Quartus Prime 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 Prime software, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus Prime 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 Prime 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 Prime 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 Prime SDC commands and are forward-annotated to the Quartus Prime 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 Prime software, refer to the Quartus Prime Help.

Related Information

- **Timing-Driven Synthesis Settings** on page 17-9
- **Quartus Prime TimeQuest Timing Analyzer Documentation**

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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime software and rerun synthesis. Altera recommends that you provide timing constraints in the Synplify software and any placement constraints in the Quartus Prime software. Repeat the process until area and timing requirements are met.

You can also use other options and techniques in the Quartus Prime software to meet area and timing requirements, such as WYSIWYG Primitive Resynthesis, which can perform optimizations on your **.vqm** netlist within the Quartus Prime 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 Prime software.

Synplify Optimization Strategies

Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus Prime 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 10-1
- **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 Prime 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 Prime software.

The physical location annotation file is called <*design name*>**_plc.tcl**. If you open the Quartus Prime software from the Synplify Premier software user interface, the Quartus Prime 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 Prime NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus Prime 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 Prime 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 Prime software. For these reasons, the Quartus Prime 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 Prime Software** on page 17-7
- **Exporting Designs to the Quartus Prime 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 Prime 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 Prime software using NativeLink integration. The Quartus Prime 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 Prime 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 implementa-tions.
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 Prime 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 Prime 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.

By default, the Synplify software generates a hierarchical **.vqm** file. To flatten the file, set the `syn_netlist_hierarchy` attribute to **0**.

Register Input and Output Delays

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 Prime 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 Prime 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 Prime 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 11-1
- **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 Prime 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 Prime project.

Verify that the correct Quartus Prime 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 Prime 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 Prime version. The Synplify software uses this information to launch the Quartus Prime software in the background. The environment variable setting must match the version of the Quartus Prime software used to generate the customized IP core.

Related Information

- **Specifying the Quartus Prime Software Version** on page 17-3
- **Using the Quartus Prime 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 Prime 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 Prime project.

Changing Synplify's Default Behavior for Instantiated Altera IP Cores

By default, the Synplify software automatically opens the Quartus Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime Placement and Routing Only** on page 17-19
- **Synplify Synthesis Techniques with the Quartus Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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 Prime 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.

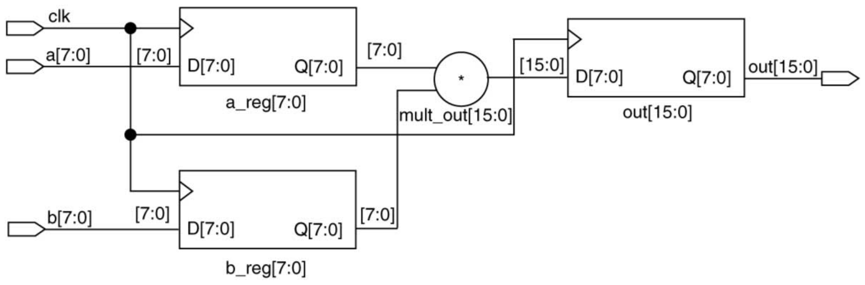
Related Information

- **Recommended HDL Coding Styles Documentation** on page 11-1

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.

Document Revision History

Table 17-4: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Optimization Settings to Compiler Settings.

Date	Version	Changes
November 2013	13.1.0	Dita conversion. Restructured content.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.
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 Prime 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 Prime 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 Prime software version 10.0 release.
November 2009	9.1.0	• Minor updates for the Quartus Prime software version 9.1 release.

Date	Version	Changes
March 2009	9.0.0	• Added new section "Exporting Designs to the Quartus Prime Software Using NativeLink Integration" on page 14–14. • Minor updates for the Quartus Prime software version 9.0 release. • Chapter 10 was previously Chapter 9 in software version 8.1.
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 Prime 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 Prime Software" • Numerous text changes and additions throughout the chapter • Renamed several sections • Updated "Referenced Documents" section

Related Information

Quartus Handbook Archive

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