Quartus Prime Pro Edition Handbook Volume 1: Design and Synthesis
Introduction to Quartus Prime Pro Edition

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

<table>
<thead>
<tr>
<th>Arria 10 Software Features</th>
<th>Quartus Prime Standard Edition</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Hybrid Placer</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>New Global Router</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>New TimeQuest</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>New Physical Synthesis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BluePrint Platform Designer</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>New Synthesis</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Rapid Recompile</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>OpenCL</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
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


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.
**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:

```bash
# 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:

```bash
# 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

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

**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
```
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

<table>
<thead>
<tr>
<th>Other Quartus Software Products</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>IP and Qsys system generation does not use proprietary Verilog configurations. The compilation library scheme changes in the following ways:</td>
</tr>
<tr>
<td></td>
<td>• 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 parameter- ization to avoid ambiguous entity instantiation errors. For example, the files for every Arria 10 PCI Express IP core variant compile into the <code>altera_pcie_a10_hip_151</code> compilation library.</td>
</tr>
<tr>
<td></td>
<td>• Simulation and synthesis file sets for IP cores and systems instantiate entities in the same manner.</td>
</tr>
<tr>
<td></td>
<td>• The generated RTL directory structure now matches the compilation library structure.</td>
</tr>
</tbody>
</table>

**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."

---

<table>
<thead>
<tr>
<th>Introduction to Quartus Prime Pro Edition</th>
<th>Altera Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Feedback</td>
<td></td>
</tr>
</tbody>
</table>
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 1-13
- Declare Objects Before Initial Values on page 1-13
- Confin e 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

<table>
<thead>
<tr>
<th>Other Quartus Software Products</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```verilog
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**

<table>
<thead>
<tr>
<th>Other Quartus Software Products</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always automatically searches your project directory and search path for undefined entities.</td>
<td>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 <code>.qsf</code> assignment: <code>set_global_assignment -name AUTO_DISCOVER_AND_SORT OFF</code></td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th><strong>Other Quartus Software Products</strong></th>
<th><strong>Quartus Prime Pro Edition</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>For the Example RTL, the analyzer searches all libraries in an unspecified order until it finds package <code>utilities_pack</code> and uses items from that package. If another library, for example <code>projectLib</code> also contains <code>utilities_pack</code>, the analyzer may use this library instead of <code>myLib.utilities_pack</code> if found before the analyzer searches <code>myLib</code>.</td>
<td>For the Example RTL, the analyzer uses the specific <code>utilities_pack</code> in <code>myLib</code>. If <code>utilities_pack</code> does not exist in library <code>myLib</code>, the analyzer generates an error.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th><strong>Other Quartus Software Products</strong></th>
<th><strong>Quartus Prime Pro Edition</strong></th>
</tr>
</thead>
</table>
| 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>
<thead>
<tr>
<th>Other Quartus Software Products</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>From the Example RTL, synthesis generates a syntax error when you specify initial values before declaring the register.</td>
</tr>
</tbody>
</table>

Example RTL:

```
1 output p_prog_io1 = p_progio1_reg;
2 reg p_progio1_reg;
```

Confine SystemVerilog Features to SystemVerilog Files


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

<table>
<thead>
<tr>
<th>Other Quartus Software Products</th>
<th>Quartus Prime Pro Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Table 1-9: ALWAYS Block Assignment Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Other Quartus Software Products</strong></td>
</tr>
<tr>
<td>Synthesis honors the mixed blocking and non-blocking assignments, although the Verilog Language Specification no longer supports this construct.</td>
</tr>
</tbody>
</table>

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 port 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>
<thead>
<tr>
<th>Table 1-10: Unconnected, Non-Existent Port Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Other Quartus Software Products</strong></td>
</tr>
<tr>
<td>Synthesis allows you to instantiate and name unconnected or non-existent ports that do not exist on the module.</td>
</tr>
</tbody>
</table>

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

---

(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."
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.

## Document Revision History

### Table 1-11: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• First version of document.</td>
</tr>
</tbody>
</table>
Managing Quartus Prime Projects

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 workflows and a scripting interface.

Quick Start

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

New Project Wizard
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

<table>
<thead>
<tr>
<th>File Type</th>
<th>Contains</th>
<th>To Edit</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project file</td>
<td>Project and revision name</td>
<td>File &gt; New Project Wizard</td>
<td>Quartus Prime Project File (.qpf)</td>
</tr>
<tr>
<td>Project settings</td>
<td>Lists design files, entity settings, target</td>
<td>Assignments &gt; Settings</td>
<td>Quartus Prime Settings File (.qsf)</td>
</tr>
<tr>
<td></td>
<td>device, synthesis directives, placement</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>constraints</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td>Clock properties, exceptions, setup/hold</td>
<td>Tools &gt; TimeQuest Timing Analyzer</td>
<td>Synopsys Design Constraints File</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
<td></td>
<td>(.sdc)</td>
</tr>
<tr>
<td>Logic design</td>
<td>RTL and other design source files</td>
<td>File &gt; New</td>
<td>All supported HDL files</td>
</tr>
<tr>
<td>files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming</td>
<td>Device programming image and information</td>
<td>Tools &gt; Programmer</td>
<td>SRAM Object File (.sof)</td>
</tr>
<tr>
<td>files</td>
<td></td>
<td></td>
<td>Programmer Object File (.pof)</td>
</tr>
<tr>
<td>Project library</td>
<td>Project and global library information</td>
<td>Tools &gt; Options &gt; Libraries</td>
<td>.qsf (project)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>quartus2.ini (global)</td>
</tr>
<tr>
<td>IP core files</td>
<td>IP core logic, synthesis, and simulation</td>
<td>Tools &gt; IP Catalog</td>
<td>All supported HDL files</td>
</tr>
<tr>
<td></td>
<td>information</td>
<td></td>
<td>Quartus Prime IP File (.qip)</td>
</tr>
<tr>
<td>Qsys system</td>
<td>Qsys system and IP core files</td>
<td>Tools &gt; Qsys</td>
<td>Qsys System File (.qsys)</td>
</tr>
<tr>
<td>files</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDA tool files</td>
<td>Generated for third-party EDA tools</td>
<td>Tools &gt; Options &gt; EDA Tool Options</td>
<td>Verilog Output File (.vo)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VHDL Output File (.vho)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Verilog Quartus Mapping File (.vqm)</td>
</tr>
</tbody>
</table>
### 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 its 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 a Combined Simulator Setup Script on page 2-36

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

**Figure 2-4: Message Suppression by Message ID Number**

**Suppressing Messages**

To suppress 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**
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

The Assignment Editor (Tools > Assignment Editor) provides a spreadsheet-like interface for assigning all instance-specific settings and constraints.
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.
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 \( f_{\text{MAX}} \) 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**

---

**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.
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 2-50

### 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 SDC Syntax Example](image)

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.

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 parameterize 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

- 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.
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**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT_DIR/*</td>
<td>Finds IP components and index files in the Quartus Prime project directory.</td>
</tr>
<tr>
<td>PROJECT_DIR/ip/**/*</td>
<td>Finds IP components and index files in any subdirectory of the /ip subdirectory of the Quartus Prime project directory.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Setting Location</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.
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**

### 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**

- **Specify your IP variation name and target device**
- **Apply preset parameters for specific applications**

---

**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>`.
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

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
<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

<IP Submodule> - IP Submodule Library

<IP Submodule> - IP Submodule Library

<IP Submodule> - IP Submodule Library

<your_ip>_tb - IP testbench system

<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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;my_ip&gt;.qsys</code></td>
<td>The Qsys system or top-level IP variation file. <code>&lt;my_ip&gt;</code> is the name that you give your IP variation.</td>
</tr>
<tr>
<td><code>&lt;system&gt;.sopcinfo</code></td>
<td>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 <code>.sopcinfo</code> file and the <code>system.h</code> 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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.cmp</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.html</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_generation.rpt</code></td>
<td>IP or Qsys generation log file. A summary of the messages during IP generation.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.debuginfo</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.qip</code></td>
<td>Contains all the required information about the IP component to integrate and compile the IP component in the Quartus Prime software.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.csv</code></td>
<td>Contains information about the upgrade status of the IP component.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.bsf</code></td>
<td>A Block Symbol File (.bsf) representation of the IP variation for use in Quartus Prime Block Diagram Files (.bdf).</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.spd</code></td>
<td>Required input file for <code>ip-make-simscript</code> to generate simulation scripts for supported simulators. The <code>.spd</code> file contains a list of files generated for simulation, along with information about memories that you can initialize.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.ppf</code></td>
<td>The Pin Planner File (.ppf) stores the port and node assignments for IP components created for use with the Pin Planner.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_bb.v</code></td>
<td>You can use the Verilog black-box (_bb.v) file as an empty module declaration for use as a black box.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_inst.v</code> or <code>&lt;my_ip&gt;_inst.vhd</code></td>
<td>HDL example instantiation template. You can copy and paste the contents of this file into your HDL file to instantiate the IP variation.</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.regmap</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.svd</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.v</code> or <code>&lt;my_ip&gt;.vhd</code></td>
<td>HDL files that instantiate each submodule or child IP core for synthesis or simulation.</td>
</tr>
<tr>
<td>mentor/</td>
<td>Contains a ModelSim® script msim_setup.tcl to set up and run a simulation.</td>
</tr>
<tr>
<td>aldec/</td>
<td>Contains a Riviera-PRO script rivierapro_setup.tcl to set up and run a simulation.</td>
</tr>
<tr>
<td>/synopsys/vcs</td>
<td>Contains a shell script vcs_setup.sh to set up and run a VCS® simulation.</td>
</tr>
<tr>
<td>/synopsys/vcsmx</td>
<td>Contains a shell script vcsmx_setup.sh and synopsys_sim.setup file to set up and run a VCS MX® simulation.</td>
</tr>
<tr>
<td>/cadence</td>
<td>Contains a shell script ncsim_setup.sh and other setup files to set up and run an NCSIM simulation.</td>
</tr>
<tr>
<td>/submodules</td>
<td>Contains HDL files for the IP core submodule.</td>
</tr>
<tr>
<td><code>&lt;IP submodule&gt;/</code></td>
<td>For each generated IP submodule directory, Qsys generates /synth and /sim sub-directories.</td>
</tr>
</tbody>
</table>
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**

### 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>_ip>` - Quartus Prime IP integration file
  - `<your_ip>_v or .vhd` - Top-level HDL 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
  - `<your_ip>_inst.v or .vhd` - Sample instantiation template
  - `<your_ip>_cmp` - VHDL component declaration file
  - `greybox_tmp`

### Generated IP File Output B

- `<Project Directory>`
  - `<your_ip>_ip>` - 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
  - `<your_ip>_vo or .vho` - IP functional simulation model
  - `<your_ip>_bsf` - Block symbol schematic file
  - `<your_ip>_html` - IP core generation report
  - `<your_ip>_testbench.v or .vhd` - Testbench file
  - `<your_ip>_instance.vo` - IP functional simulation model
  - `<your_ip>_instance.vhd` - HDL synthesis files
  - `<your_ip>_testbench.v` - Testbench file
  - `<your_ip>_library` - Contains IP subcomponent synthesis libraries

### Generated IP File Output C

- `<Project Directory>`
  - `<your_ip>_ip>` - Quartus Prime IP integration file
  - `<your_ip>_v or .vhd` - Top-level HDL synthesis file
  - `<your_ip>_inst.v or .vhd` - Sample instantiation template
  - `<your_ip>_cmp` - VHDL component declaration file
  - `<your_ip>_syn.v or .vhd` - Timing & resource estimation netlist
  - `<your_ip>_inst.v or .vhd` - Sample instantiation template
  - `<your_ip>_cmp` - VHDL component declaration file
  - `<your_ip>_syn.v or .vhd` - Timing & resource estimation netlist
  - `<your_ip>_inst.v or .vhd` - Sample instantiation template

### Generated IP File Output D

- `<Project Directory>`
  - `<your_ip>_ip>` - Quartus Prime IP integration file
  - `<your_ip>_v or .vhd` - Top-level HDL synthesis file
  - `<your_ip>_inst.v or .vhd` - Sample instantiation template
  - `<your_ip>_bb` - 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>_testbench.v` - Testbench file
  - `<your_ip>_library` - Contains IP subcomponent synthesis libraries

#### 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.
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 2-5: `qsys-generate` Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;1st arg file&gt;</code></td>
<td>Required</td>
<td>The name of the <code>.qsys</code> system file to generate.</td>
</tr>
<tr>
<td>`--synthesis=&lt;VERILOG</td>
<td>VHDL&gt;`</td>
<td>Optional</td>
</tr>
<tr>
<td><code>--block-symbol-file</code></td>
<td>Optional</td>
<td>Creates a Block Symbol File (.bsf) for the Qsys system.</td>
</tr>
<tr>
<td>`--simulation=&lt;VERILOG</td>
<td>VHDL&gt;`</td>
<td>Optional</td>
</tr>
<tr>
<td>`--testbench=&lt;SIMPLE</td>
<td>STANDARD&gt;`</td>
<td>Optional</td>
</tr>
<tr>
<td>`--testbench-simulation=&lt;VERILOG</td>
<td>VHDL&gt;`</td>
<td>Optional</td>
</tr>
<tr>
<td><strong>Option</strong></td>
<td><strong>Usage</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>--search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>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 &quot;$&quot;, for example, &quot;/extra/dir,$&quot;.</td>
</tr>
<tr>
<td><code>--jvm-max-heap-size=&lt;value&gt;</code></td>
<td>Optional</td>
<td>The maximum memory size that Qsys uses for allocations when running <code>qsys-generate</code>. You specify the value as <code>&lt;size&gt;&lt;unit&gt;</code>, where <code>unit</code> is m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512m.</td>
</tr>
<tr>
<td><code>--family=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Specifies the device family.</td>
</tr>
<tr>
<td><code>--part=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Specifies the device part number. If set, this option overrides the <code>--family</code> option.</td>
</tr>
<tr>
<td><code>--allow-mixed-language-simulation</code></td>
<td>Optional</td>
<td>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 <code>--file-set=&lt;value&gt;</code> for all components for simulation model generation. The current version of the ModelSim-Altera simulator supports mixed language simulation.</td>
</tr>
</tbody>
</table>

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

### Modifying an IP Variation

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

**Table 2-6: Modifying an IP Variation**

<table>
<thead>
<tr>
<th><strong>Menu Command</strong></th>
<th><strong>Action</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File &gt; Open</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>View &gt; Utility Windows &gt; Project Navigator &gt; IP Components</strong></td>
<td>Double-click the IP variation to launch the parameter editor and modify the IP variation. Regenerate the IP variation to implement your changes.</td>
</tr>
</tbody>
</table>
Menu Command | Action
--- | ---
Project > Upgrade IP Components | Select the IP variation and click Upgrade in Editor to launch the parameter editor and modify the IP variation. Regenerate the IP variation to implement your changes.

### Upgrading IP Cores

IP core variants generated with a previous version 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**

<table>
<thead>
<tr>
<th>Instance</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arria 10: 10AX1154F4835GES</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>IP Core Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Upgraded</td>
<td>Your IP variation uses the latest version of the IP core.</td>
</tr>
<tr>
<td>IP Upgrade Optional</td>
<td>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.</td>
</tr>
<tr>
<td>IP Upgrade Required</td>
<td>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.</td>
</tr>
<tr>
<td>IP Upgrade Unsupported</td>
<td>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.</td>
</tr>
<tr>
<td>IP End of Life</td>
<td>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.</td>
</tr>
<tr>
<td>IP Upgrade Mismatch Warning</td>
<td>Warning of non-critical IP core differences in migrating IP to another device family.</td>
</tr>
</tbody>
</table>

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

![Image of Upgrade IP Components dialog box]

**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:

  ```bash
  quartus_sh -ip_upgrade -variation_files <my_ip>.<qsys,.v, .vhd> <quartus_project>
  ```

  Example:
  ```bash
  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:

```bash
quartus_sh -ip_upgrade -variation_files "<my_ip1>.qsys,.v,.vhd>;<my_ip_filepath/my_ip2>.hdl" <quartus_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**
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

<table>
<thead>
<tr>
<th>Upgrade IP Components Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regeneration Status</td>
<td>Displays the &quot;Success&quot; or &quot;Failed&quot; status of each upgrade or migration. Click the status of any failed upgrade to open a detailed <strong>IP Upgrade Report</strong>.</td>
</tr>
<tr>
<td>Version</td>
<td>Dynamically updates to the new version number when upgrade is successful. The text is red when upgrade is required.</td>
</tr>
<tr>
<td>Device Family</td>
<td>Dynamically updates to the new device family when migration is successful. The text is red when upgrade is required.</td>
</tr>
<tr>
<td>Description</td>
<td>Summarizes IP release information and displays actionable, corrective action for resolving upgrade or migration failures. Follow these instructions to resolve upgrade failures. Click the <strong>Release Notes</strong> link for the latest known issues about the Altera IP core.</td>
</tr>
<tr>
<td>Perform Automatic Upgrade</td>
<td>Runs automatic upgrade on all IP cores that support auto upgrade. Also, automatically generates a <code>&lt;Project Directory&gt;/ip_upgrade_port_diff_report</code> 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.</td>
</tr>
</tbody>
</table>

### Figure 2-22: Resolving Upgrade Errors

![Image of Upgrade Details]

- **Upgrade details**
  - **Upgrade success**
    - `my_sw_v1re4` and `Virtex v14.1` with version `14.1`
  - `my_sw_v1re4` and `Virtex v14.1` with version `15.1`
- **Upgrade failed (click to open report)**
  - `my_sw_v1re4` and `SE4 II` with version `15.1`
- **Upgrade manually**
  - `my_sw_v1re4` and `Virtex v14.1`
  - `my_sw_v1re4` and `SE4 II`

*Error*: IP was converted to use new infrastructure but failed. 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` commands and use 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**

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**

The Quartus Prime software supports RTL and gate-level simulation of Altera IP cores in supported EDA simulators. The Quartus Prime software generates simulation files for each IP core during IP generation, including the functional simulation model, any testbench (or example design), and vendor-specific simulator setup scripts for each IP core. You can use the functional simulation model and the testbench or example design generated with your IP core for simulation. The IP generation output also includes scripts to compile and run any testbench. The generated scripts list all models or libraries required to simulate your IP core.

The Quartus Prime software provides integration with your simulator and supports multiple simulation flows, including your own scripted and custom simulation flows. Whichever flow you chose, IP core simulation involves the following steps:

1. Generate simulation model, testbench (or example design), and simulator setup script files
2. Set up your simulator environment and any simulation script(s)
3. Compile simulation model libraries
4. Run your simulator

The Quartus Prime software integrates with your preferred simulation environment. This section describes how to setup and run typical scripted and NativeLink simulation flows.

**Related Information**

**Simulating Altera Designs**

**Generating IP Simulation Files**

The Quartus Prime software optionally generates the functional simulation model, any testbench (or example design), and vendor-specific simulator setup scripts when you generate the IP core from the parameter editor or command-line including for each IP core. Use the following to control the generation of IP simulation files:

- Click **Assignment > Settings** to specify your supported simulator and options for IP simulation file generation.
- Click **Tools > IP Catalog** to parameterize a new IP variation, enable generation of simulation files, and generate the IP core synthesis and simulation files.
- Click **View > Utility Windows > Project Navigator > IP Components** to edit parameters and regenerate synthesis or simulation for an existing IP core variation.
### Table 2-9: Altera IP Simulation Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator setup scripts</td>
<td>Vendor-specific scripts to compile, elaborate, and simulate Altera IP models</td>
<td>&lt;my_dir&gt;/aldec/rivierapro_setup.tcl</td>
</tr>
<tr>
<td></td>
<td>and simulation model library files. Source these files from your top-level</td>
<td>&lt;my_dir&gt;/cadence/ncsim_setup.sh</td>
</tr>
<tr>
<td></td>
<td>simulation script, or edit these files to compile, elaborate, and simulate</td>
<td>&lt;my_dir&gt;/mentor/msim_setup.tcl</td>
</tr>
<tr>
<td></td>
<td>your design and testbench.</td>
<td>&lt;my_dir&gt;/synopsys/vcs/vcs_setup.sh</td>
</tr>
<tr>
<td>Simulation IP File</td>
<td>Contains IP core simulation library mapping information. To use NativeLink,</td>
<td>&lt;design name&gt;.sip</td>
</tr>
<tr>
<td></td>
<td>you must add the .qip and .sip files generated for IP or Qsys systems to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>your project.</td>
<td></td>
</tr>
<tr>
<td>Qsys System File</td>
<td>Contains IP core simulation library mapping information. To use NativeLink</td>
<td>&lt;design name&gt;.sip</td>
</tr>
<tr>
<td></td>
<td>for Arria 10 devices and later, you must add the .qsys file generated for IP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or Qsys system to your project.</td>
<td></td>
</tr>
<tr>
<td>IP functional simulation models</td>
<td>IP functional simulation models are cycle-accurate VHDL or Verilog HDL</td>
<td>&lt;my_ip&gt;.vho</td>
</tr>
<tr>
<td></td>
<td>models generated by the Quartus Prime software for some Altera IP cores. IP</td>
<td>&lt;my_ip&gt;.vo</td>
</tr>
<tr>
<td></td>
<td>functional simulation models support fast functional simulation of IP using</td>
<td></td>
</tr>
<tr>
<td></td>
<td>industry-standard VHDL and Verilog HDL simulators.</td>
<td></td>
</tr>
<tr>
<td>IEEE encrypted models</td>
<td>Arria V, Cyclone V, Stratix V, and newer simulation model libraries and IP</td>
<td>&lt;my_ip&gt;.v</td>
</tr>
<tr>
<td></td>
<td>simulation models are provided in Verilog HDL and IEEE encrypted Verilog HDL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VHDL simulation of these models is supported using your simulator's</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co-simulation capabilities. IEEE encrypted Verilog HDL models are</td>
<td></td>
</tr>
<tr>
<td></td>
<td>significantly faster than IP functional simulation models.</td>
<td></td>
</tr>
</tbody>
</table>

**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.

### Scripting IP Simulation

The Quartus Prime software supports the use of scripts to automate simulation processing in your preferred simulation environment. You can use your preferred scripting methodology to control simulation.

Altera recommends the use of a version-independent top-level simulation script to control design, testbench, and IP core simulation. Because Quartus Prime-generated simulation file names may change...
after IP upgrade or regeneration, Altera recommends that your top-level simulation script "sources" any generated setup scripts, rather than using the generated setup scripts directly. You can use the `ip-setup-simulation` utility to generate or regenerate underlying setup scripts after any software or IP version upgrade or regeneration. Use of a top-level script and `ip-setup-simulation` eliminates the requirement to manually update simulation scripts.

Figure 2-24: Incorporating Generated Simulator Setup Scripts into a Top-Level Simulation Script

Generating a Combined Simulator Setup Script

The Quartus Prime software provides utilities to help you generate and update IP simulation scripts. You can use the `ip-setup-simulation` utility to generate a combined simulator setup script, for all Altera IP in your design, for each supported simulator. You can subsequently rerun `ip-setup-simulation` to automatically update the combined script. Each simulator's combined script file contains a rudimentary template that you can adapt for integration of the setup script into a top-level simulation script.

Table 2-10: Simulation Script Utilities

<table>
<thead>
<tr>
<th>Utility</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ip-setup-simulation</code></td>
<td>Generates a combined, version-independent simulation script for all Altera IP cores in your project, and automates regeneration of the script after upgrading software or IP versions. Use the <code>--compile-to-work</code> option to compile all simulation files into a single work library if your simulation environment requires that structure. Use the <code>--use-relative-paths</code> option to use relative paths whenever possible.</td>
</tr>
<tr>
<td></td>
<td><code>ip-setup-simulation</code></td>
</tr>
<tr>
<td></td>
<td><code>--quartus-project=&lt;my_proj&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>--output-directory=&lt;my_dir&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>--use-relative-paths</code></td>
</tr>
<tr>
<td></td>
<td><code>--compile-to-work</code></td>
</tr>
<tr>
<td></td>
<td><code>--use-relative-paths</code> and <code>--compile-to-work</code> are optional. For command-line help listing all options for these executables, type: <code>&lt;utility name&gt; --help</code>.</td>
</tr>
<tr>
<td><code>ip-make-simscript</code></td>
<td>Generates a combined simulation script for all IP cores specified on the command line. Specify one or more <code>.spd</code> files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries.</td>
</tr>
<tr>
<td></td>
<td><code>ip-make-simscript</code></td>
</tr>
<tr>
<td></td>
<td><code>--spd=&lt;ipA.spd,ipB.spd&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>--output-directory=&lt;directory&gt;</code></td>
</tr>
</tbody>
</table>
Running ip-setup-simulation

To generate or update combined simulator setup scripts for all IP cores in your design, follow these steps:

1. Generate, regenerate, or upgrade one or more Altera IP core.
2. Run `ip-setup-simulation` on the project containing the IP core:

   ```
   ip-setup-simulation --quartus-project=<my proj>.qpf
   --output-directory=<my dir>
   --use-relative-paths
   ```

3. To incorporate the simulator setup script into your top-level simulation script, refer to the template section in the generated simulator setup script as a guide to sourcing the generated script:
   a. Copy the specified template sections from the simulator-specific generated scripts and paste them into a new top-level file.
   b. Remove the comments at the beginning of each line from the copied template sections.
   c. Include the customizations required to match your design simulation requirements, for example:
      - Specify the `TOP_LEVEL_NAME` variable to the design’s simulation top-level file. 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.
      - If necessary, set the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files.
      - Compile the top-level HDL file (e.g. a test program) and all other files in the design.
      - 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.

4. To automatically update the combined IP simulation scripts, run `ip-setup-simulation` after any of the following events:
   - IP core initial generation or regeneration with new parameters
   - Upgrade of Quartus Prime software version
   - Upgrade of IP core version

Refer to the following topics for detailed steps for using the templates for each vendor.

Sourcing Aldec Simulator Setup 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
   ```
Sourcing Cadence Simulator Setup 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`.

```bash
# # 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
```

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

```bash
# Start of template
# If the copied and modified template file is "aldec.do", run it as:
# vsim -c -do aldec.do
# # Compile the 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:

```bash
set TOP_LEVEL_NAME sim_top
vlog -sv2k5 ..../sim_top.sv
```

4. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

5. Run the new top-level script from the generated simulation directory:

```bash
vsim -c -do <path to sim_top>.tcl
```
2. Delete the first two characters of each line (comment and space):

```bash
# 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_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:

```bash
TOP_LEVEL_NAME=sim_top \
ncvlog -sv "$QSYS_SIMDIR/../top.sv"
```

4. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

5. Run the resulting top-level script from the generated simulation directory by specifying the path to `ncsim.sh`.

**Sourcing ModelSim Simulator Setup 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`.

```bash
# # 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
```
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. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

5. Run the resulting top-level script from the generated simulation directory:

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

### Sourcing VCS Simulator Setup 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. If necessary, add the QSYS_SIMDIR variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

6. Run the resulting top-level script from the generated simulation directory by specifying the path to vcs_sim.sh.

Sourcing VCS MX Simulator Setup 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 \

Send Feedback
Using NativeLink Simulation

The NativeLink feature integrates your EDA simulator with the Quartus Prime software and automates the following simulation steps:

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

```bash
# 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 \nSKIP_SIM=1 \

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

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 your top-level file, for example:

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

5. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

6. Run the resulting top-level script from the generated simulation directory by specifying the path to `vcsmx_sim.sh`.
• Set and reuse simulation settings
• Generate simulator-specific files and simulation scripts
• Compile Altera simulation libraries
• Launch your simulator automatically following Quartus Prime Analysis & Elaboration, Analysis & Synthesis, or after a full compilation.

Note: The Quartus Prime Pro Edition software does not support NativeLink RTL simulation. To use NativeLink for Arria 10 devices, you must add the .qsys file generated for the IP or Qsys system. To use NativeLink for all other device families, you must add to your project the .qip and .sip files generated for IP or Qsys systems.

Setting Up NativeLink Simulation
Before running simulation using the NativeLink flow, you must specify settings for your simulator in the Quartus Prime software. To specify simulation settings in the Quartus Prime software, follow these steps:

1. Open a Quartus Prime project.
2. Click Tools > Options and specify the location of your simulator executable file.

Table 2-11: Execution Paths for EDA Simulators

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mentor Graphics ModelSim-Altera</td>
<td><code>&lt;drive letter&gt;:\&lt;simulator install path&gt;\win32aloem</code> (Windows)</td>
</tr>
<tr>
<td></td>
<td><code>\&lt;simulator install path&gt;\bin</code> (Linux)</td>
</tr>
<tr>
<td>Mentor Graphics ModelSim</td>
<td><code>&lt;drive letter&gt;:\&lt;simulator install path&gt;\win32</code> (Windows)</td>
</tr>
<tr>
<td>Mentor Graphics QuestaSim</td>
<td><code>&lt;simulator install path&gt;\bin</code> (Linux)</td>
</tr>
<tr>
<td>Synopsys VCS/VCS MX</td>
<td><code>&lt;simulator install path&gt;\bin</code> (Linux)</td>
</tr>
<tr>
<td>Cadence Incisive Enterprise</td>
<td><code>&lt;simulator install path&gt;\tools\bin</code> (Linux)</td>
</tr>
<tr>
<td>Aldec Active-HDL Aldec Riviera-PRO</td>
<td><code>&lt;drive letter&gt;:\&lt;simulator install path&gt;\bin</code> (Windows)</td>
</tr>
<tr>
<td></td>
<td><code>&lt;simulator install path&gt;\bin</code> (Linux)</td>
</tr>
</tbody>
</table>

3. Click Assignments > Settings and specify options on the Simulation page and More NativeLink Settings dialog box. Specify default options for simulation library compilation, netlist and tool command script generation, and for launching RTL or gate-level simulation automatically following Quartus Prime processing.
4. If your design includes a testbench, turn on Compile test bench and then click Test Benches to specify options for each testbench. Alternatively, turn on Use script to compile testbench and specify the script file.
5. If you want to use a script to setup simulation, turn on Use script to setup simulation.
Generating IP Functional Simulation Models for 40nm Devices

Altera provides IP functional simulation models for some Altera IP cores supporting 40nm Altera devices. To generate IP functional simulation models, follow these steps:

- Turn on the Generate Simulation Model option when parameterizing the IP core.
- When you simulate your design, compile only the .vo or .vho for these IP cores in your simulator. In this case you should not compile the corresponding HDL file. The encrypted HDL file supports synthesis by only the Quartus Prime software.

Note: Altera IP cores that do not require IP functional simulation models for simulation, do not provide the Generate Simulation Model option in the IP core parameter editor.

Note: Many recently released Altera IP cores support RTL simulation using IEEE Verilog HDL encryption. IEEE encrypted models are significantly faster than IP functional simulation models. You can simulate the models in both Verilog HDL and VHDL designs.

Related Information
AN 343: OpenCore Evaluation of AMPP Megafuncions

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.

```verilog
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));
```
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 files for board-level verification (Assignments > Settings > EDA Tool Settings).
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

- **Mentor Graphics Precision Synthesis Support** on page 16-1
- **Simulating Altera Designs**

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.
Related Information

- [link/mwh1409960181641/mwh1409958516629](link/mwh1409960181641/mwh1409958516629)
- [link/mwh1409960181641/mwh1409959857762](link/mwh1409960181641/mwh1409959857762)

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**.
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.vhd.
- Ensure that all the subdirectories are in the same hierarchical structure and relative path as in the original platform.

Figure 2-27: 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.
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_SCRIPT_FILE <file name>.
   ```

### Project Revision Commands

Use the following commands for scripting project revisions.

- **Create Revision Command** on page 2-51
- **Set Current Revision Command** on page 2-51
- **Get Project Revisions Command** on page 2-51
- **Delete Revision Command** on page 2-51

#### Create Revision Command

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>based_on</td>
<td>(optional) Specifies the revision name on which the new revision bases its settings.</td>
</tr>
<tr>
<td>set_current</td>
<td>(optional) Sets the new revision as the current revision.</td>
</tr>
</tbody>
</table>

#### 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.

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

#### Get Project Revisions Command

```tcl
get_project_revisions <project_name>
```

#### Delete Revision Command

```tcl
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 2-52

**Report Specified Project Libraries Commands**
on page 2-52

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 2-12: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016.02.09</td>
<td>15.1.1</td>
<td>• Clarified instructions for Generating a Combined Simulator Setup Script.</td>
</tr>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Added Generating Version-Independent IP Simulation Scripts topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added example IP simulation script templates for supported simulators.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Incorporating IP Simulation Scripts in Top-Level Scripts topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Troubleshooting IP Upgrade topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated IP Catalog and parameter editor descriptions for GUI changes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated IP upgrade and migration steps for latest GUI changes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Generating IP Cores process for GUI changes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Files Generated for IP Cores and Qsys system description.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed references to devices and features not supported in version 15.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>• Added description of design templates feature.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated screenshot for DSE II GUI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added <code>qsys_script</code> IP core instantiation information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Described changes to generating and processing of instance and entity names.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added description of upgrading IP cores at the command line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated procedures for upgrading and migrating IP cores.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Gate level timing simulation supported only for Cyclone IV and Stratix IV devices.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>• Updated content for DSE II GUI and optimizations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about new <em>Assignments &gt; Settings &gt; IP Settings</em> that control frequency of synthesis file regeneration and automatic addition of IP files to the project.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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

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 often 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

<table>
<thead>
<tr>
<th>Design Planning Factor</th>
<th>SignapT II Logic Analyzer</th>
<th>System Console</th>
<th>In-System Memory Content Editor</th>
<th>Logic Analyzer Interfac e (LAI)</th>
<th>SignalProbe</th>
<th>In-System Sources and Probes</th>
<th>Virtual JTAG IP Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG connections</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Additional logic resources</td>
<td>—</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>Reserve device memory</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Reserve I/O pins</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Instantiate IP core in your HDL code</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2015.11.02 | 15.1.0  | • Added references to BluePrint Design Planning chapter.  
<p>|            |         | • 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.             |</p>
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November, 2012</td>
<td>12.1.0</td>
<td>Update for changes to early pin planning feature</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Editorial update.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>• Added link to System Design with Qsys in “Creating Design Specifications” on page 1–2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Simultaneous Switching Noise Analysis” on page 1–8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Planning for On-Chip Debugging Tools” on page 1–10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed information from “Planning Design Partitions and Floorplan Location Assignments” on page 1–15</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “System Design and Standard Interfaces” on page 1–3 to include information about the Qsys system integration tool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added link to the Altera Product Selector in “Device Selection” on page 1–3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Converted information into new table (Table 1–1) in “Planning for On-Chip Debugging Options” on page 1–10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Simplified description of incremental compilation usages in “Incremental Compilation with Design Partitions” on page 1–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about the Rapid Recompile option in “Flat Compilation Flow with No Design Partitions” on page 1–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed details and linked to Quartus Prime Help in “Fast Synthesis and Early Timing Estimation” on page 1–16</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Added new section “System Design” on page 1–3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed details about debugging tools from “Planning for On-Chip Debugging Options” on page 1–10 and referred to other handbook</td>
</tr>
<tr>
<td></td>
<td></td>
<td>chapters for more information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated information on recommended design flows in “Incremental Compilation with Design Partitions” on page 1–14 and removed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Single-Project Versus Multiple-Project Incremental Flows” heading</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Merged the “Planning Design Partitions” section with the “Creating a Design Floorplan” section. Changed heading title to “Planning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Design Partitions and Floorplan Location Assignments” on page 1–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed “Creating a Design Floorplan” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed “Referenced Documents” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor updates throughout chapter</td>
</tr>
</tbody>
</table>
### 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.
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

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory-Mapped</td>
<td>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 unidirectional (read only or write only), or bi-directional (read and write).</td>
</tr>
<tr>
<td>Streaming</td>
<td>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.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>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</td>
</tr>
<tr>
<td>Clocks</td>
<td>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.</td>
</tr>
<tr>
<td>Resets</td>
<td>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.</td>
</tr>
<tr>
<td>Conduits</td>
<td>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.</td>
</tr>
</tbody>
</table>
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]

**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**

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**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT_DIR/*</td>
<td>Finds IP components and index files in the Quartus Prime project directory.</td>
<td></td>
</tr>
<tr>
<td>PROJECT_DIR/ip/**/*</td>
<td>Finds IP components and index files in any subdirectory of the ip subdirectory of the Quartus Prime project directory.</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Setting Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tools &gt; Options &gt; IP Settings</strong>&lt;br&gt;Or&lt;br&gt;Assignments &gt; Settings &gt; IP Settings (only enabled with open project)</td>
<td>• Specify your IP <em>generation HDL preference</em>. The parameter editor generates IP files in your preferred HDL by default.&lt;br&gt;• Increase Maximum Qsys memory usage size if you experience slow processing for large systems, or if Qsys reports an Out of Memory error.&lt;br&gt;• Specify whether to <strong>Automatically add Quartus Prime IP files</strong> 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.&lt;br&gt;• Use the <strong>IP Regeneration Policy</strong> setting to control when synthesis files regenerate for each IP variation. Typically you <strong>Always regenerate synthesis files for IP cores</strong> after making changes to an IP variation.</td>
</tr>
<tr>
<td><strong>Tools &gt; Options &gt; IP Catalog Search Locations</strong>&lt;br&gt;Or&lt;br&gt;Assignments &gt; Settings &gt; IP Catalog Search Locations</td>
<td>• 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.</td>
</tr>
</tbody>
</table>

### 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 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**

```xml
<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**

```xml
<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 preset 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

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 \( f_{\text{Max}} \) 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.

### 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 $f_{\text{MAX}}$. 
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.
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.
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. For example, you can import or export a layout from or to a different directory.
The shortcut, **Ctrl-3**, for example, allows you to quickly change your Qsys window view with a quick keystroke.

![Manage Custom Layouts](image)

**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.
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**

---

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.
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 multiple 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.
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.

---

**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
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 interface 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**
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 Check 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.
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:

```tcl
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:

```tcl
set_module_property COMPOSITION_CALLBACK <name of callback procedure>
```

Specify the appropriate Tcl commands inside the Tcl procedure with the following syntax:

```tcl
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.

```tcl
# 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**

```plaintext
get\_parameter\_value <parameter>
```

**Returns**

The value of the parameter.

**Arguments**

**parameter**

The name of the parameter whose value is being retrieved.

**Example**

```plaintext
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
   - Simulation at Unit-Level, Possibly Using BFMs
     - Does Simulation Give Expected Results?
       - Yes
       - No
         - Debug Design
   - Complete System, Add and Connect All IP Components, Define Memory Map If Needed
     - Generate Qsys System
     - Does Simulation Give Expected Results?
       - Yes
       - No
         - Debug Design
     - Perform System-Level Simulation
     - Constrain, Compile in Quartus Prime Generating .sof
       - Download .sof to PCB with FPGA
         - Does HW Testing Give Expected Results?
           - Yes
           - Qsys System Complete
           - No
             - 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.
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 upgrades 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

<table>
<thead>
<tr>
<th>Upgrade IP Components Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regeneration Status</td>
<td>Displays the &quot;Success&quot; or &quot;Failed&quot; status of each upgrade or migration. Click the status of any failed upgrade to open a detailed IP Upgrade Report.</td>
</tr>
<tr>
<td>Version</td>
<td>Dynamically updates to the new version number when upgrade is successful. The text is red when upgrade is required.</td>
</tr>
<tr>
<td>Device Family</td>
<td>Dynamically updates to the new device family when migration is successful. The text is red when upgrade is required.</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Perform Automatic Upgrade</td>
<td>Runs automatic upgrade on all IP cores that support auto upgrade. Also, automatically generates a <code>&lt;Project Directory&gt;/ip_upgrade_port_diff_report</code> 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.</td>
</tr>
</tbody>
</table>

**Figure 4-14: Resolving Upgrade Errors**

![Image of Upgrade Errors Resolution](image-url)
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

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.
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 subsystem 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**

---

**Edit a Qsys Subsystem**

You can double-click a Qsys subsystem in the **Hierarchy** tab to edit its contents in any tab. When you make a change, open tabs refresh their content to reflect your edit. You can change the level of a subsystem, or push it into another subsystem with commands in the **System Contents** tab.

**Note:** To edit a .qsys file, the file must be writeable and reside outside of the ACDS installation directory. You cannot edit systems that you create from composed _hw.tcl files, or systems that define instance parameters.

1. In the **System Contents** or **Schematic** tabs, use the hierarchy widget to navigate to the top-level system, up one level, or down one level (drill into a system).
All tabs refresh and display the requested hierarchy level.

2. To edit a system, double-click the system in the **Hierarchy** tab. You can also drill into the system with the Hierarchy tool or right-click commands, which are available in the **Hierarchy, Schematic, System Contents** tabs.

   The system is open and available for edit in all Qsys views. A system currently open for edit appears as bold in the **Hierarchy** tab.

3. In the **System Contents** tab, you can rename any element, add, remove, or duplicate connections, and export interfaces, as appropriate.

   Changes to a subsystem affect all instances. Qsys identifies unsaved changes to a subsystem with an asterisk next to the subsystem in the **Hierarchy** tab.

**Related Information**

*View a Schematic of Your Qsys System* on page 4-16

**Change the Hierarchy Level of a Qsys Component**

You can push selected components down into their own subsystem, 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.
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.
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.

```plaintext
# 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.
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**.
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 parameters for the `memory_system.qsys` display in the parameter editor.
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` are displayed.
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**

**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 enable 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

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

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.
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**

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**

---

**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.
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.

---

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit interconnect pipeline</td>
<td>Specifies the maximum number of pipeline stages that Qsys may insert in each command and response path to increase the ( f_{\text{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 <strong>Memory-Mapped Interconnect</strong> tab. Access this tab by clicking <strong>Show System With Qsys Interconnect command</strong> on the <strong>System</strong> menu.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| **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 $f_{\text{max}}$, 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 $f_{\text{max}}$, but higher area. This setting is useful when you have AXI masters or slaves and you want a higher $f_{\text{max}}$.  |
| **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 $f_{\text{max}}$, 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 $f_{\text{max}}$, but higher area. This setting is useful when you have AXI masters or slaves and you want a higher $f_{\text{max}}$.  |
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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>• Non-secure</td>
<td>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. <strong>Note:</strong> You can also set these values in the <strong>Security</strong> column in the <strong>System Contents</strong> tab.</td>
</tr>
<tr>
<td></td>
<td>• Secure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Secure ranges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TrustZone-aware</td>
<td></td>
</tr>
<tr>
<td>Secure address ranges</td>
<td>Accepts valid address range.</td>
<td>Allows you to type in any valid address range.</td>
</tr>
</tbody>
</table>


**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](#)
- [Qsys System Design Components](#)

### 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.

<table>
<thead>
<tr>
<th>Compile-Time Security Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure</td>
<td>Master sends only non-secure transactions, and the slave receives any transaction, secure or non-secure.</td>
</tr>
<tr>
<td>Secure</td>
<td>Master sends only secure transactions, and the slave receives only secure transactions.</td>
</tr>
<tr>
<td>Secure ranges</td>
<td>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, <code>0x0:0xfff, 0x2000:0x20ff</code>.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>TrustZone-aware Master</th>
<th>Non-TrustZone-aware Master Secure</th>
<th>Non-TrustZone-aware Master Non-Secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrustZone-aware slave/memory</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Non-TrustZone-aware slave (secure)</td>
<td>Per-access</td>
<td>OK</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Non-TrustZone-aware slave (non-secure)</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Non-TrustZone-aware memory (secure region)</td>
<td>Per-access</td>
<td>OK</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>
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 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**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum Qsys memory usage</strong></td>
<td>Allows you to increase memory usage for Qsys if you experience slow processing for large systems, or if Qsys reports an <strong>Out of Memory</strong> error.</td>
</tr>
<tr>
<td><strong>IP generation HDL preference</strong></td>
<td>The Quartus Prime software uses this setting when the <code>.qsys</code> file appears in the <strong>Files</strong> list for the current project in the <strong>Settings</strong> dialog box and you run Analysis &amp; Synthesis. Qsys uses this setting when you generate HDL files.</td>
</tr>
<tr>
<td><strong>Automatically add Quartus Prime IP files to all projects</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>IP Catalog Search Locations</strong></td>
<td>The Quartus Prime software uses the settings that you specify for global and project search paths under <strong>IP Search Locations</strong>, in addition to the <strong>IP Search Path</strong> in Qsys (<strong>Tools &gt; Options</strong>), to populate the Quartus Prime software IP Catalog. Qsys uses the uses the settings that you specify for global search paths under <strong>IP Search Locations</strong> to populate the Qsys IP Catalog, which appears in Qsys (<strong>Tools &gt; Options</strong>). Qsys uses the project search path settings to populate the Qsys IP Catalog when you open Qsys from within the Quartus Prime software (<strong>Tools &gt; Qsys</strong>), but not when you open Qsys from the command-line.</td>
</tr>
</tbody>
</table>

**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 gigabyte 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:

```plaintext
create_clock -name master_clk -period 20 [get_ports {clk}]
derive_pll_clocks
```

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create HDL design files for synthesis</td>
<td>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.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Create timing and resource estimates for third-party EDA synthesis tools</strong></td>
<td>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 <code>&lt;your_ip_component_name&gt;_syn.v</code>.</td>
</tr>
<tr>
<td><strong>Create Block Symbol File (.bsf)</strong></td>
<td>Allows you to optionally create a (.bsf) file to use in a schematic Block Diagram File (.bdf).</td>
</tr>
<tr>
<td><strong>Create simulation model</strong></td>
<td>Allows you to optionally generate Verilog HDL or VHDL simulation model files, and simulation scripts.</td>
</tr>
<tr>
<td><strong>Allow mixed-language simulation</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Clear output directories for selected generation targets</strong></td>
<td>Clears previous generation attempts for current synthesis or simulation.</td>
</tr>
</tbody>
</table>

**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>_inst.v` or `.vhd` - Lists file for IP core synthesis

- `<your_ip>.qip` - Lists files for IP core synthesis

- `<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>.sopcinfo` - Software tool-chain integration file

- `<your_ip>_bb.v` - Verilog HDL black box EDA synthesis file

- `<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>_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
  - `<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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;my_ip&gt;.qsys</code></td>
<td>The Qsys system or top-level IP variation file. <code>&lt;my_ip&gt;</code> is the name that you give your IP variation.</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;system&gt;.sopcinfo</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.cmp</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.html</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_generation.rpt</code></td>
<td>IP or Qsys generation log file. A summary of the messages during IP generation.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.debuginfo</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.qip</code></td>
<td>Contains all the required information about the IP component to integrate and compile the IP component in the Quartus Prime software.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.csv</code></td>
<td>Contains information about the upgrade status of the IP component.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.bsf</code></td>
<td>A Block Symbol File (.bsf) representation of the IP variation for use in Quartus Prime Block Diagram Files (.bdf).</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.spd</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.ppf</code></td>
<td>The Pin Planner File (.ppf) stores the port and node assignments for IP components created for use with the Pin Planner.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_bb.v</code></td>
<td>You can use the Verilog black-box (_bb.v) file as an empty module declaration for use as a black box.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;_inst.v</code> or <code>&lt;my_ip&gt;_inst.vhd</code></td>
<td>HDL example instantiation template. You can copy and paste the contents of this file into your HDL file to instantiate the IP variation.</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.regmap</code></td>
<td>If the IP contains register information, the <code>.regmap</code> file generates. The <code>.regmap</code> file describes the register map information of master and slave interfaces. This file complements the <code>.sopinfo</code> file by providing more detailed register information about the system. This enables register display views and user customizable statistics in System Console.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.svd</code></td>
<td>Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Qsys system. During synthesis, the <code>.svd</code> files for slave interfaces visible to System Console masters are stored in the <code>.sof</code> 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.</td>
</tr>
<tr>
<td><code>&lt;my_ip&gt;.v</code> or <code>&lt;my_ip&gt;.vhd</code></td>
<td>HDL files that instantiate each submodule or child IP core for synthesis or simulation.</td>
</tr>
<tr>
<td><code>mentor/</code></td>
<td>Contains a ModelSim script <code>msim_setup.tcl</code> to set up and run a simulation.</td>
</tr>
<tr>
<td><code>aldec/</code></td>
<td>Contains a Riviera-PRO script <code>rivierapro_setup.tcl</code> to setup and run a simulation.</td>
</tr>
<tr>
<td><code>/synopsys/vcs</code></td>
<td>Contains a shell script <code>vcs_setup.sh</code> to set up and run a VCS simulation.</td>
</tr>
<tr>
<td><code>/synopsys/vcsmx</code></td>
<td>Contains a shell script <code>vcsmx_setup.sh</code> and <code>synopsys_sim.setup</code> file to set up and run a VCS MX simulation.</td>
</tr>
<tr>
<td><code>/cadence</code></td>
<td>Contains a shell script <code>ncsim_setup.sh</code> and other setup files to set up and run an NCSIM simulation.</td>
</tr>
<tr>
<td><code>/submodules</code></td>
<td>Contains HDL files for the IP core submodule.</td>
</tr>
<tr>
<td><code>&lt;IP submodule&gt;/</code></td>
<td>For each generated IP submodule directory, Qsys generates <code>/synth</code> and <code>/sim</code> sub-directories.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 its 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**

<table>
<thead>
<tr>
<th>File Name or Directory Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;system&gt;_tb.qsys</code></td>
<td>The Qsys testbench system.</td>
</tr>
<tr>
<td>File Name or Directory Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>&lt;system&gt;_tb.v</code> or <code>&lt;system&gt;_tb.vhd</code></td>
<td>The top-level testbench file that connects BFMs to the top-level interfaces of <code>&lt;system&gt;_tb.qsys</code>.</td>
</tr>
<tr>
<td><code>&lt;system&gt;_tb.spd</code></td>
<td>Required input file for <code>ip-make-simscript</code> to generate simulation scripts for supported simulators. The <code>.spd</code> file contains a list of files generated for simulation and information about memory that you can initialize.</td>
</tr>
<tr>
<td><code>&lt;system&gt;.html</code> and <code>&lt;system&gt;_tb.html</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>&lt;system&gt;_generation.rpt</code></td>
<td>Qsys generation log file. A summary of the messages that Qsys issues during testbench system generation.</td>
</tr>
<tr>
<td><code>&lt;system&gt;.ipx</code></td>
<td>The IP Index File (.ipx) lists the available IP components, or a reference to other directories to search for IP components.</td>
</tr>
<tr>
<td><code>&lt;system&gt;.svd</code></td>
<td>Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Qsys system. Similarly, during synthesis the <code>.svd</code> files for slave interfaces visible to System Console masters are stored in the <code>.sof</code> 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.</td>
</tr>
<tr>
<td><code>mentor/</code></td>
<td>Contains a ModelSim script <code>msim_setup.tcl</code> to set up and run a simulation</td>
</tr>
<tr>
<td><code>aldec/</code></td>
<td>Contains a Riviera-PRO script <code>rivierapro_setup.tcl</code> to setup and run a simulation.</td>
</tr>
<tr>
<td><code>/synopsys/vcs</code></td>
<td>Contains a shell script <code>vcs_setup.sh</code> to set up and run a VCS simulation.</td>
</tr>
<tr>
<td><code>/synopsys/vcsmx</code></td>
<td>Contains a shell script <code>vcsmx_setup.sh</code> and <code>synopsys_simsetup</code> file to set up and run a VCS MX simulation.</td>
</tr>
<tr>
<td><code>/cadence</code></td>
<td>Contains a shell script <code>ncsim_setup.sh</code> and other setup files to set up and run an NCSIM simulation.</td>
</tr>
<tr>
<td><code>/submodules</code></td>
<td>Contains HDL files for the submodule of the Qsys testbench system.</td>
</tr>
<tr>
<td><code>&lt;child IP cores&gt;/</code></td>
<td>For each generated child IP core directory, Qsys testbench generates <code>/synth</code> and <code>/sim</code> subdirectories.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP_LEVEL_NAME</td>
<td>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.</td>
</tr>
<tr>
<td>QSYS_SIMDIR</td>
<td>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.</td>
</tr>
<tr>
<td>QUARTUS_INSTALL_DIR</td>
<td>Points to the Quartus installation directory that contains the device family library.</td>
</tr>
</tbody>
</table>

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.

```verilog
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 a Combined Simulator Setup Script
The Quartus Prime software provides utilities to help you generate and update IP simulation scripts. You can use the `ip-setup-simulation` utility to generate a combined simulator setup script, for all Altera IP in your design, for each supported simulator. You can subsequently rerun `ip-setup-simulation` to automatically update the combined script. Each simulator's combined script file contains a rudimentary template that you can adapt for integration of the setup script into a top-level simulation script.

Table 4-13: Simulation Script Utilities

<table>
<thead>
<tr>
<th>Utility</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ip-setup-simulation</code></td>
<td>Generates a combined, version-independent simulation script for all Altera IP cores in your project, and automates regeneration of the script after upgrading software or IP versions. Use the <code>compile-to-work</code> option to compile all simulation files into a single work library if your simulation environment requires that structure. Use the <code>--use-relative-paths</code> option to use relative paths whenever possible.</td>
</tr>
<tr>
<td><code>ip-make-simscript</code></td>
<td>Generates a combined simulation script for all IP cores specified on the command line. Specify one or more <code>.spd</code> files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries.</td>
</tr>
</tbody>
</table>

Running `ip-setup-simulation`
To generate or update combined simulator setup scripts for all IP cores in your design, follow these steps:

1. Generate, regenerate, or upgrade one or more Altera IP core.
2. Run `ip-setup-simulation` on the project containing the IP core:

   ```
   ip-setup-simulation --quartus-project=<my proj>.qpf
   --output-directory=<my_dir>
   --use-relative-paths
   --compile-to-work
   ```

3. To incorporate the simulator setup script into your top-level simulation script, refer to the template section in the generated simulator setup script as a guide to sourcing the generated script:
a. Copy the specified template sections from the simulator-specific generated scripts and paste them into a new top-level file.
b. Remove the comments at the beginning of each line from the copied template sections.
c. Include the customizations required to match your design simulation requirements, for example:
   - Specify the TOP_LEVEL_NAME variable to the design’s simulation top-level file. 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.
   - If necessary, set the QSYS_SIMDIR variable to point to the location of the generated IP simulation files.
   - Compile the top-level HDL file (e.g. a test program) and all other files in the design.
   - 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.

4. To automatically update the combined IP simulation scripts, run ip-setup-simulation after any of the following events:
   - IP core initial generation or regeneration with new parameters
   - Upgrade of Quartus Prime software version
   - Upgrade of IP core version

Refer to the following topics for detailed steps for using the templates for each vendor.

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.

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 pipeline 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]| □
3.004  0.204  mm_domain_0|addr_router_001|Equal5~0|data\box
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. 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:

```bash
<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:

```bash
<Quartus Prime installation directory>/quartus/sopc_builder/bin/<executable name> --help
```

**Example 4-5: Qsys Command-Line Scripting Example**

```tcl
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1st arg file</code></td>
<td>Optional</td>
<td>The name of the <code>.qsys</code> system or <code>.qvar</code> variation file to edit.</td>
</tr>
<tr>
<td><code>--search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>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 &quot;$&quot;, for example: /extra/dir.$</td>
</tr>
<tr>
<td><code>--project-directory=&lt;directory&gt;</code></td>
<td>Optional</td>
<td>Allows you to find IP components in certain locations relative to the project, if any. By default, the current directory is: <code>.‘. To exclude any project directory, use </code>' .`</td>
</tr>
<tr>
<td><code>--new-component-type=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Allows you to specify the kind of instance that is parameterized in a variation.</td>
</tr>
<tr>
<td><code>--debug</code></td>
<td>Optional</td>
<td>Enables debugging features and output.</td>
</tr>
<tr>
<td><code>--host-controller</code></td>
<td>Optional</td>
<td>Launches the application with an XML host controller interface on standard input/output.</td>
</tr>
<tr>
<td><code>--jvm-max-heap-size=&lt;value&gt;</code></td>
<td>Optional</td>
<td>The maximum memory size Qsys uses for allocations when running qsys-edit. You specify this value as <code>&lt;size&gt;&lt;unit&gt;</code>, where unit is m (or M) for multiples of megabytes, or g (or G) for multiples of gigabytes. The default value is 512m.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Optional</td>
<td>Display help for qsys-edit.</td>
</tr>
</tbody>
</table>

### 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:

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

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

   ```bash
   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

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1st arg file&gt;</td>
<td>Required</td>
<td>The name of the .qsys system file to generate.</td>
</tr>
<tr>
<td>--synthesis=&lt;VERILOG</td>
<td>VHDL&gt;</td>
<td>Optional</td>
</tr>
<tr>
<td>--block-symbol-file</td>
<td>Optional</td>
<td>Creates a Block Symbol File (.bsf) for the Qsys system.</td>
</tr>
<tr>
<td>--simulation=&lt;VERILOG</td>
<td>VHDL&gt;</td>
<td>Optional</td>
</tr>
<tr>
<td>--testbench=&lt;SIMPLE</td>
<td>STANDARD&gt;</td>
<td>Optional</td>
</tr>
<tr>
<td>--testbench-simulation=&lt;VERILOG</td>
<td>VHDL&gt;</td>
<td>Optional</td>
</tr>
<tr>
<td>--search-path=&lt;value&gt;</td>
<td>Optional</td>
<td>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 &quot;$&quot;, for example, &quot;/extra/dir,$&quot;.</td>
</tr>
<tr>
<td>--jvm-max-heap-size=&lt;value&gt;</td>
<td>Optional</td>
<td>The maximum memory size that Qsys uses for allocations when running qsys-generate. You specify the value as &lt;size&gt;&lt;unit&gt;, where unit is m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512m.</td>
</tr>
</tbody>
</table>
### 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:

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

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--project-dir=&lt;directory&gt;</code></td>
<td>Optional</td>
<td>Finds IP components relative to the Quartus Prime project directory. By default, Qsys uses <code>.</code> as the current directory. To exclude a project directory, leave the value empty.</td>
</tr>
<tr>
<td><code>--name=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Provides a pattern to filter the names of the IP components found. To show all IP components, use <code>*</code> or <code>''</code>. By default, Qsys shows all IP components. The argument is not case sensitive.</td>
</tr>
<tr>
<td><code>--verbose</code></td>
<td>Optional</td>
<td>Reports the progress of the command.</td>
</tr>
<tr>
<td><code>--xml</code></td>
<td>Optional</td>
<td>Generates the output in XML format, in place of colon-delimited format.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--source-directory=&lt;directory&gt;</td>
<td>Optional</td>
<td>Specifies the root directory for IP component files. The default directory is <em>.</em>. You can provide a comma-separated list of directories.</td>
</tr>
<tr>
<td>--output=&lt;file&gt;</td>
<td>Optional</td>
<td>Specifies the name of the file to generate. The default name is /component.ipx.</td>
</tr>
<tr>
<td>--relative-vars=&lt;value&gt;</td>
<td>Optional</td>
<td>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.</td>
</tr>
<tr>
<td>--thorough-descent</td>
<td>Optional</td>
<td>If set, a component or .ipx file in a directory does not stop Qsys from searching subdirectories.</td>
</tr>
<tr>
<td>--message-before=&lt;value&gt;</td>
<td>Optional</td>
<td>Prints a message: stdout when indexing begins.</td>
</tr>
<tr>
<td>--message-after=&lt;value&gt;</td>
<td>Optional</td>
<td>Sends a message: stdout when indexing is done.</td>
</tr>
<tr>
<td>--help</td>
<td>Optional</td>
<td>Displays help for the ip-make-ipx command.</td>
</tr>
</tbody>
</table>

Related Information

*Set up the IP Index File (.ipx) to Search for IP Components* on page 4-5

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--system-file=&lt;file&gt;</code></td>
<td>Optional</td>
<td>Specifies the path to a <code>.qsys</code> file. Qsys loads the system before running scripting commands.</td>
</tr>
<tr>
<td><code>--script=&lt;file&gt;</code></td>
<td>Optional</td>
<td>A file that contains Tcl scripting commands that you can use to create or manipulate Qsys systems. If you specify both <code>--cmd</code> and <code>--script</code>, Qsys runs the <code>--cmd</code> commands before the script specified by <code>--script</code>.</td>
</tr>
<tr>
<td><code>--cmd=&lt;value&gt;</code></td>
<td>Optional</td>
<td>A string that contains Tcl scripting commands that you can use to create or manipulate a Qsys system. If you specify both <code>--cmd</code> and <code>--script</code>, Qsys runs the <code>--cmd</code> commands before the script specified by <code>--script</code>.</td>
</tr>
<tr>
<td><code>--package-version=&lt;value&gt;</code></td>
<td>Optional</td>
<td>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 <code>package require -exact qsys &lt;version&gt;</code> command.</td>
</tr>
<tr>
<td><code>--search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>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 &quot;$&quot;, for example, /&lt;directory path&gt;/dir,$. Separate multiple directory references with a comma.</td>
</tr>
<tr>
<td><code>--jvm-max-heap-size=&lt;value&gt;</code></td>
<td>Optional</td>
<td>The maximum memory size that the <code>qsys-script</code> tool uses. You specify this value as <code>&lt;size&gt;&lt;unit&gt;</code>, where unit is m (or M) for multiples of megabytes, or g (or G) for multiples of gigabytes.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Optional</td>
<td>Displays help for the <code>qsys-script</code> utility.</td>
</tr>
</tbody>
</table>
Qsys Scripting Command Reference

The following are Qsys scripting commands:

- `add_connection` on page 4-83
- `add_instance` on page 4-84
- `add_interface` on page 4-85
- `apply_preset` on page 4-86
- `auto_assign_base_addresses` on page 4-87
- `auto_assign_system_base_addresses` on page 4-88
- `auto_assign_irqs` on page 4-89
- `auto_connect` on page 4-90
- `create_system` on page 4-91
- `export_hw_tcl` on page 4-92
- `get_composed_connection_parameter_value` on page 4-93
- `get_composed_connection_parameters` on page 4-94
- `get_composed_connections` on page 4-95
- `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_composed_instances` on page 4-100
- `get_connection_parameter_property` on page 4-101
- `get_connection_parameter_value` on page 4-102
- `get_connection_parameters` on page 4-103
- `get_connection_properties` on page 4-104
- `get_connection_property` on page 4-105
- `get_connections` on page 4-106
- `get_instance_assignment` on page 4-107
- `get_instance_assignments` on page 4-108
- `get_instance_documentation_links` on page 4-109
- `get_instance_interface_assignment` on page 4-110
- `get_instance_interface_assignments` on page 4-111
- `get_instance_interface_parameter_property` on page 4-112
- `get_instance_interface_parameter_value` on page 4-113
- `get_instance_interface_parameters` on page 4-114
set_instance_parameter_value on page 4-149
set_instance_property on page 4-150
set_interconnect_requirement on page 4-151
set_interface_property on page 4-152
set_module_property on page 4-153
set_project_property on page 4-154
set_use_testbench_naming_pattern on page 4-155
set_validation_property on page 4-156
unlock_avalon_base_address on page 4-157
validate_connection on page 4-158
validate_instance on page 4-159
validate_instance_interface on page 4-160
validate_system on page 4-161
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 appear under the **System** category in the IP Category.

**Usage**

```tcl
export_hw_tcl
```

**Returns**

No return value.

**Arguments**

No arguments

**Example**

```tcl
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**

```plaintext
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**

```plaintext
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**
```plaintext
get_composed_instance_assignments <instance> <child_instance>
```

**Returns**
A list of assignment names.

**Arguments**
- `instance`
The child instance containing a subsystem.
- `child_instance`
The name of a child instance found in the subsystem.

**Example**
```plaintext
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 parameter 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**
```plaintext
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**
```plaintext
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 parameter names.

Arguments
connection
The connection to query.

Example
get_connection_parameters cpu.data_master/dma0.csr

Related Information
• get_connection_parameter_property on page 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**

```text
get_instance_properties
```

**Returns**

A list of property names for the child instance.

**Arguments**

No arguments.

**Example**

```text
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:

```bash
foreach { element_id name value } $requirement_list { loop_body }
```

Arguments

No arguments.

Example

```bash
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 of 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
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 on page 4-163
Design Environment Type Properties on page 4-164
Direction Properties on page 4-165
Element Properties on page 4-166
Instance Properties on page 4-167
Interface Properties on page 4-168
Message Levels Properties on page 4-169
Module Properties on page 4-170
Parameter Properties on page 4-171
Parameter Status Properties on page 4-174
Parameter Type Properties on page 4-175
Port Properties on page 4-176
Project Properties on page 4-177
System Info Type Properties on page 4-178
Units Properties on page 4-180
Validation Properties on page 4-181
### Connection Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>END</td>
<td>The end interface of the connection.</td>
</tr>
<tr>
<td>string</td>
<td>NAME</td>
<td>The name of the connection.</td>
</tr>
<tr>
<td>string</td>
<td>START</td>
<td>The start interface of the connection.</td>
</tr>
<tr>
<td>String</td>
<td>TYPE</td>
<td>The type of the connection.</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE</td>
<td>The design environment supports native IP interfaces.</td>
</tr>
<tr>
<td>QSYS</td>
<td>The design environment supports standard Qsys interfaces.</td>
</tr>
</tbody>
</table>
## Direction Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDIR</td>
<td>The direction for a bidirectional signal.</td>
</tr>
<tr>
<td>INOUT</td>
<td>The direction for an input signal.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>The direction for an output signal.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>AUTHOR</td>
<td>The author of the component or interface.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AUTO_EXPORT</td>
<td>Indicates whether unconnected interfaces on the instance are automatically exported.</td>
</tr>
<tr>
<td>String</td>
<td>CLASS_NAME</td>
<td>The type of the instance, interface or connection, for example, altera_nios2 or avalon_slave.</td>
</tr>
<tr>
<td>String</td>
<td>DESCRIPTION</td>
<td>The description of the instance, interface or connection type.</td>
</tr>
<tr>
<td>String</td>
<td>DISPLAY_NAME</td>
<td>The display name for referencing the type of instance, interface or connection.</td>
</tr>
<tr>
<td>Boolean</td>
<td>EDITABLE</td>
<td>Indicates whether you can edit the component in the Qsys Component Editor.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>Indicates whether the instance is turned on.</td>
</tr>
<tr>
<td>String</td>
<td>GROUP</td>
<td>The IP Catalog category.</td>
</tr>
<tr>
<td>Boolean</td>
<td>INTERNAL</td>
<td>Hides internal IP components or sub-components from the IP Catalog.</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
<td>The name of the instance, interface or connection.</td>
</tr>
<tr>
<td>String</td>
<td>VERSION</td>
<td>The version number of the instance, interface or connection, for example, 14.0.</td>
</tr>
</tbody>
</table>
### Instance Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>AUTO_EXPORT</td>
<td>Indicates whether unconnected interfaces on the instance are automatically exported.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>If true, this instance is included in the generated system. if false, it is not included.</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
<td>The name of the system, which is used as the name of the top-level module in the generated HDL.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>String</td>
<td>EXPORT_OF</td>
<td>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: <code>&lt;instanceName.interfaceName&gt;</code>. For example:</td>
</tr>
</tbody>
</table>

```
set_interface_property CSC_input EXPORT_OF my_colorSpace-Converter.input_port
```
### Message Levels Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPONENT_INFO</td>
<td>Reports an informational message only during component editing.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Provides messages when debug mode is turned on.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Provides an error message.</td>
</tr>
<tr>
<td>INFO</td>
<td>Provides an informational message.</td>
</tr>
<tr>
<td>PROGRESS</td>
<td>Reports progress during generation.</td>
</tr>
<tr>
<td>TODOERROR</td>
<td>Provides an error message that indicates the system is incomplete.</td>
</tr>
<tr>
<td>WARNING</td>
<td>Provides a warning message.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>String</td>
<td>GENERATION_ID</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
</tr>
</tbody>
</table>
## Parameter Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>AFFECTS_ELABORATION</td>
<td>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.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AFFECTS_GENERATION</td>
<td>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.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AFFECTS_VALIDATION</td>
<td>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.</td>
</tr>
<tr>
<td>String[]</td>
<td>ALLOWED_RANGES</td>
<td>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,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOWED_RANGES {&quot;dev1:Cyclone IV GX&quot;,&quot;dev2:Stratix V GT&quot;}</td>
</tr>
<tr>
<td>String</td>
<td>DEFAULT_VALUE</td>
<td>The default value.</td>
</tr>
<tr>
<td>Boolean</td>
<td>DERIVED</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>DESCRIPTION</td>
<td>A short user-visible description of the parameter, suitable for a tooltip description in the parameter editor.</td>
</tr>
<tr>
<td>String[]</td>
<td>DISPLAY_HINT</td>
<td>Provides a hint about how to display a property.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>boolean</td>
<td>DISPLAY_NAME</td>
<td>The GUI label that appears to the left of this parameter.</td>
</tr>
<tr>
<td>String</td>
<td>DISPLAY_UNITS</td>
<td>The GUI label that appears to the right of the parameter.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>When False, the parameter is turned off. It displays in the parameter editor but is grayed out, indicating that you cannot edit this parameter.</td>
</tr>
<tr>
<td>String</td>
<td>GROUP</td>
<td>Controls the layout of parameters in the GUI.</td>
</tr>
<tr>
<td>Boolean</td>
<td>HDL_PARAMETER</td>
<td>When True, Qsys passes the parameter to the HDL component description. The default value is False.</td>
</tr>
<tr>
<td>String</td>
<td>LONG_DESCRIPTION</td>
<td>A user-visible description of the parameter. Similar to DESCRIPTION, but allows a more detailed explanation.</td>
</tr>
<tr>
<td>String</td>
<td>NEW_INSTANCE_VALUE</td>
<td>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.</td>
</tr>
<tr>
<td>String[]</td>
<td>SYSTEM_INFO</td>
<td>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,</td>
</tr>
<tr>
<td>String</td>
<td>SYSTEM_INFO_ARG</td>
<td>Defines an argument to pass to SYSTEM_INFO. For example, the name of a reset interface.</td>
</tr>
<tr>
<td>(various)</td>
<td>SYSTEM_INFO_TYPE</td>
<td>Specifies the types of system information that you can query. Refer to System Info Type Properties.</td>
</tr>
<tr>
<td>(various)</td>
<td>TYPE</td>
<td>Specifies the type of the parameter. Refer to Parameter Type Properties.</td>
</tr>
<tr>
<td>(various)</td>
<td>UNITS</td>
<td>Sets the units of the parameter. Refer to Units Properties.</td>
</tr>
<tr>
<td>Boolean</td>
<td>VISIBLE</td>
<td>Indicates whether or not to display the parameter in the parameter editor.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>String</td>
<td>WIDTH</td>
<td>Indicates the width of the logic vector for the <code>STD_LOGIC_VECTOR</code> parameter.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>ACTIVE</td>
<td>Indicates that this parameter is an active parameter.</td>
</tr>
<tr>
<td>Boolean</td>
<td>DEPRECATED</td>
<td>Indicates that this parameter exists only for backwards compatibility, and may not have any effect.</td>
</tr>
<tr>
<td>Boolean</td>
<td>EXPERIMENTAL</td>
<td>Indicates that this parameter is experimental and not exposed in the design flow.</td>
</tr>
</tbody>
</table>
### Parameter Type Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>A boolean parameter set to <code>true</code> or <code>false</code>.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>A signed 32-bit floating point parameter. (Not supported for HDL parameters.)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>A signed 32-bit integer parameter.</td>
</tr>
<tr>
<td>INTEGER_LIST</td>
<td>A parameter that contains a list of 32-bit integers. (Not supported for HDL parameters.)</td>
</tr>
<tr>
<td>LONG</td>
<td>A signed 64-bit integer parameter. (Not supported for HDL parameters.)</td>
</tr>
<tr>
<td>NATURAL</td>
<td>A 32-bit number that contains values 0 to 2147483647 (0x7fffffff).</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>A 32-bit number that contains values 1 to 2147483647 (0x7fffffff).</td>
</tr>
<tr>
<td>STD_LOGIC</td>
<td>A single bit parameter set to 0 or 1.</td>
</tr>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>An arbitrary-width number. The parameter property <code>WIDTH</code> determines the size of the logic vector.</td>
</tr>
<tr>
<td>STRING</td>
<td>A string parameter.</td>
</tr>
<tr>
<td>STRING_LIST</td>
<td>A parameter that contains a list of strings. (Not supported for HDL parameters.)</td>
</tr>
</tbody>
</table>
## Port Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(various)</td>
<td>DIRECTION</td>
<td>The direction of the signal. Refer to <em>Direction Properties</em>.</td>
</tr>
<tr>
<td>String</td>
<td>ROLE</td>
<td>The type of the signal. Each interface type defines a set of interface types for its ports.</td>
</tr>
<tr>
<td>Integer</td>
<td>WIDTH</td>
<td>The width of the signal in bits.</td>
</tr>
</tbody>
</table>
## Project Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>DEVICE</td>
<td>The device part number in the Quartus Prime project that contains the Qsys system.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FAMILY</td>
<td>The device family name in the Quartus Prime project that contains the Qsys system.</td>
</tr>
</tbody>
</table>
### System Info Type Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>ADDRESS_MAP</td>
<td>An XML-formatted string that describes the address map for the interface specified in the <code>SYSTEM_INFO</code> parameter property.</td>
</tr>
<tr>
<td>Integer</td>
<td>ADDRESS_WIDTH</td>
<td>The number of address bits that Qsys requires to address memory-mapped slaves connected to the specified memory-mapped master in this instance.</td>
</tr>
<tr>
<td>String</td>
<td>AVALON_SPEC</td>
<td>The version of the Qsys interconnect. Refer to <em>Avalon Interface Specifications</em>.</td>
</tr>
<tr>
<td>Integer</td>
<td>CLOCK_DOMAIN</td>
<td>An integer that represents the clock domain for the interface specified in the <code>SYSTEM_INFO</code> 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.</td>
</tr>
<tr>
<td>Long, Integer</td>
<td>CLOCK_RATE</td>
<td>The rate of the clock connected to the clock input specified in the <code>SYSTEM_INFO</code> parameter property. If zero, the clock rate is currently unknown.</td>
</tr>
<tr>
<td>String</td>
<td>CLOCK_RESET_INFO</td>
<td>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 <em>Avalon Interface Specifications</em>.</td>
</tr>
<tr>
<td>String</td>
<td>CUSTOM_INSTRUCTION_SLAVES</td>
<td>Provides slave information, including the name, base address, address span, and clock cycle type.</td>
</tr>
<tr>
<td>String</td>
<td>DESIGN_ENVIRONMENT</td>
<td>A string that identifies the current design environment. Refer to <em>Design Environment Type Properties</em>.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE</td>
<td>The device part number of the selected device.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FAMILY</td>
<td>The family name of the selected device.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FEATURES</td>
<td>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 <code>array</code> command. The keys are device features. The values are 1 if the feature is present, and 0 if the feature is absent.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_SPEEDGRADE</td>
<td>The speed grade of the selected device.</td>
</tr>
<tr>
<td>Integer</td>
<td>GENERATION_ID</td>
<td>A integer that stores a hash of the generation time that Qsys uses as a unique ID for a generation run.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BigInteger, Long</td>
<td>INTERRUPTS_USED</td>
<td>A mask indicating which bits of an interrupt receiver are connected to interrupt senders. The interrupt receiver is specified in the system info argument.</td>
</tr>
<tr>
<td>Integer</td>
<td>MAX_SLAVE_DATA_WIDTH</td>
<td>The data width of the widest slave connected to the specified memory-mapped master.</td>
</tr>
<tr>
<td>String, Boolean, Integer</td>
<td>QUARTUS_INI</td>
<td>The value of the <code>quartus.ini</code> setting specified in the system info argument.</td>
</tr>
<tr>
<td>Integer</td>
<td>RESET_DOMAIN</td>
<td>An integer representing the reset domain for the interface specified in the <code>SYSTEM_INFO</code> 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.</td>
</tr>
<tr>
<td>String</td>
<td>TRISTATECONDUIT_INFO</td>
<td>An XML description of the tri-state conduit masters connected to a tri-state conduit slave. The slave is specified as the <code>SYSTEM_INFO</code> parameter property. The value contains information about the slave, connected master instance and interface names, and signal names, directions, and widths.</td>
</tr>
<tr>
<td>String</td>
<td>TRISTATECONDUIT_MASTERS</td>
<td>The names of the instance's interfaces that are tri-state conduit slaves.</td>
</tr>
<tr>
<td>String</td>
<td>UNIQUE_ID</td>
<td>A string guaranteed to be unique to this instance.</td>
</tr>
</tbody>
</table>

**Related Information**

- [Design Environment Type Properties](#) on page 4-164
- [Avalon Interface Specifications](#)
- [Qsys Interconnect](#) on page 6-1
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>A memory-mapped address.</td>
</tr>
<tr>
<td>BITS</td>
<td>Memory size in bits.</td>
</tr>
<tr>
<td>BITSPERSECOND</td>
<td>Rate in bits per second.</td>
</tr>
<tr>
<td>BYTES</td>
<td>Memory size in bytes.</td>
</tr>
<tr>
<td>CYCLES</td>
<td>A latency or count in clock cycles.</td>
</tr>
<tr>
<td>GIGABITSPERSECOND</td>
<td>Rate in gigabits per second.</td>
</tr>
<tr>
<td>GIGABYTES</td>
<td>Memory size in gigabytes.</td>
</tr>
<tr>
<td>GIGAHERTZ</td>
<td>Frequency in GHz.</td>
</tr>
<tr>
<td>HERTZ</td>
<td>Frequency in Hz.</td>
</tr>
<tr>
<td>KILOBITSPERSECOND</td>
<td>Rate in kilobits per second.</td>
</tr>
<tr>
<td>KILOBYTES</td>
<td>Memory size in kilobytes.</td>
</tr>
<tr>
<td>KILOHERTZ</td>
<td>Frequency in kHz.</td>
</tr>
<tr>
<td>MEGABITSPERSECOND</td>
<td>Rate, in megabits per second.</td>
</tr>
<tr>
<td>MEGABYTES</td>
<td>Memory size in megabytes.</td>
</tr>
<tr>
<td>MEGAHERTZ</td>
<td>Frequency in MHz.</td>
</tr>
<tr>
<td>MICROSECONDS</td>
<td>Time in microseconds.</td>
</tr>
<tr>
<td>MILLISECONDS</td>
<td>Time in milliseconds.</td>
</tr>
<tr>
<td>NANOSECONDS</td>
<td>Time in nanoseconds.</td>
</tr>
<tr>
<td>NONE</td>
<td>Unspecified units.</td>
</tr>
<tr>
<td>PERCENT</td>
<td>A percentage.</td>
</tr>
<tr>
<td>PICOSECONDS</td>
<td>Time in picoseconds.</td>
</tr>
<tr>
<td>SECONDS</td>
<td>Time in seconds.</td>
</tr>
</tbody>
</table>
### Validation Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>AUTOMATIC_VALIDATION</td>
<td>When <code>true</code>, Qsys runs system validation and elaboration after each scripting command. When <code>false</code>, 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.</td>
</tr>
</tbody>
</table>
# 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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Added: <em>Troubleshooting IP or Qsys System Upgrade</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <em>Generating Version-Agnostic IP and Qsys Simulation Scripts</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>• New figure: <em>Avalon-MM Write Master Timing Waveforms in the Parameters Tab</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added <em>Enable ECC protection</em> option, <em>Specify Qsys Interconnect Requirements</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added External Memory Interface Debug Toolkit note, <em>Generate a Qsys System</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Modelsim-Altera now supports native mixed-language (VHDL/Verilog/SystemVerilog) simulation, <em>Generating Files for Synthesis and Simulation</em>.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• Create and Manage Hierarchical Qsys Systems.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Schematic tab.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• View and Filter Clock and Reset Domains.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <em>File &gt; Recent Projects</em> menu item.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated example: Hierarchical System Using Instance Parameters.</td>
</tr>
<tr>
<td>August 2014</td>
<td>14.0a10.0</td>
<td>• Added distinction between legacy and standard device generation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated: <em>Upgrading Outdated IP Components</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated: <em>Generating a Qsys System</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated: <em>Integrating a Qsys System with the Quartus Prime Software</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added screen shot: <em>Displaying Your Qsys System</em>.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Added tab descriptions: Details, Connections.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added <em>Managing IP Settings in the Quartus Prime Software</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added <em>Upgrading Outdated IP Components</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added <em>Support for Avalon-MM Non-Power of Two Data Widths</em>.</td>
</tr>
</tbody>
</table>
### Changes

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>• Added Integrating with the .qsys File.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Using the Hierarchy Tab.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Managing Interconnect Requirements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Viewing Qsys Interconnect.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>• Added AMBA APB support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added qsys-generate utility.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added VHDL BFM ID support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Creating Secure Systems (TrustZones).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added CMSIS Support for Qsys Systems With An HPS Component.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added VHDL language support options.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Added AMBA AXI4 support.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Added AMBA AX3I support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Preset Editor updates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added command-line utilities, and scripts.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>• Added Synopsys VCS and VCS MX Simulation Shell Script.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Cadence Incisive Enterprise (NCSIM) Simulation Shell Script.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Using Instance Parameters and Example Hierarchical System Using Parameters.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>• Added simulation support in Verilog HDL and VHDL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added testbench generation support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated simulation and file generation sections.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the Quartus Prime Handbook, refer to the Quartus Handbook Archive.
Creating Qsys Components

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

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory-Mapped</td>
<td>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 unidirectional (read only or write only), or bi-directional (read and write).</td>
</tr>
<tr>
<td>Streaming</td>
<td>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.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>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</td>
</tr>
<tr>
<td>Clocks</td>
<td>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.</td>
</tr>
<tr>
<td>Resets</td>
<td>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.</td>
</tr>
<tr>
<td>Conduits</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 Layer Tool 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.
Specify IP Component Type Information

- **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.

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.

```tcl
# 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:/alera/15.0/avalon_mm_simple_slave_template.v

HDL Language: Verilog

HDL Preview:

```v
// avalon_mm_simple_slave_template.v

// This file was auto-generated as a prototype implementation of a module created in component editor. It ties off all outputs to ground and ignores all inputs. It needs to be edited to make it do something useful.
//
// This file will not be automatically regenerated. You should check 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
    );
```
If you already have 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 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.

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

```tcl
# 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
```
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.

<table>
<thead>
<tr>
<th>Interface Prefix</th>
<th>Interface Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>asi</td>
<td>Avalon-ST sink (input)</td>
</tr>
<tr>
<td>aso</td>
<td>Avalon-ST source (output)</td>
</tr>
<tr>
<td>avm</td>
<td>Avalon-MM master</td>
</tr>
<tr>
<td>avs</td>
<td>Avalon-MM slave</td>
</tr>
<tr>
<td>axm</td>
<td>AXI master</td>
</tr>
<tr>
<td>axs</td>
<td>AXI slave</td>
</tr>
<tr>
<td>apm</td>
<td>APB master</td>
</tr>
<tr>
<td>aps</td>
<td>APB slave</td>
</tr>
<tr>
<td>coe</td>
<td>Conduit</td>
</tr>
<tr>
<td>csi</td>
<td>Clock Sink (input)</td>
</tr>
<tr>
<td>Interface Prefix</td>
<td>Interface Type</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>cso</td>
<td>Clock Source (output)</td>
</tr>
<tr>
<td>inr</td>
<td>Interrupt receiver</td>
</tr>
<tr>
<td>ins</td>
<td>Interrupt sender</td>
</tr>
<tr>
<td>ncm</td>
<td>Nios II custom instruction master</td>
</tr>
<tr>
<td>ncs</td>
<td>Nios II custom instruction slave</td>
</tr>
<tr>
<td>rsi</td>
<td>Reset sink (input)</td>
</tr>
<tr>
<td>rso</td>
<td>Reset source (output)</td>
</tr>
<tr>
<td>tcm</td>
<td>Avalon-TC master</td>
</tr>
<tr>
<td>tcs</td>
<td>Avalon-TC slave</td>
</tr>
</tbody>
</table>

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.
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

```tcl
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
```
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_VARIABLES 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 \( \{\text{width}\} \) 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\}"
```

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.

Related Information

- Component Interface Tcl Reference on page 8-1
- CMSIS - Cortex Microcontroller Software

Creating Qsys Components
6. To move signals between interfaces, select the signal, and then drag it to another interface.
7. To rename a signal 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

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.
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.

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: \(<n-1>:0\).

```tcl
# 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"
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:

\[
\text{maximum_master_id_width_in_the_interconnect} + \log_2(\text{number_of_masters_in_the_same_interconnect})
\]

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:

\[
5 \text{ bits} + 2 \text{ bits} = 7
\]

**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.

<table>
<thead>
<tr>
<th>AXI Master Parameters</th>
<th>AXI Slave Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>readIssuingCapability</td>
<td>readAcceptanceCapability</td>
</tr>
<tr>
<td>writeIssuingCapability</td>
<td>writeAcceptanceCapability</td>
</tr>
<tr>
<td>combinedIssuingCapability</td>
<td>combinedAcceptanceCapability</td>
</tr>
<tr>
<td></td>
<td>readDataReorderingDepth</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>ALLOWED_RANGES Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{a b c}</code></td>
<td>a, b, or c</td>
</tr>
<tr>
<td><code>{&quot;No Control&quot; &quot;Single Control&quot; &quot;Dual Controls&quot;}</code></td>
<td>Unique string values. Quotation marks are required if the strings include spaces.</td>
</tr>
<tr>
<td><code>{1 2 4 8 16}</code></td>
<td>1, 2, 4, 8, or 16</td>
</tr>
<tr>
<td><code>{1:3}</code></td>
<td>1 through 3, inclusive.</td>
</tr>
<tr>
<td><code>{1 2 3 7:10}</code></td>
<td>1, 2, 3, or 7 through 10 inclusive.</td>
</tr>
</tbody>
</table>

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.

```tcl
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 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"
```

...
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.

```tcl
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

- Control Interfaces Dynamically with an Elaboration Callback on page 5-26
- Component Interface Tcl Reference on page 8-1
Control Interfaces Dynamically with an Elaboration Callback

You can allow user parameters to dynamically control your component's behavior with 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

```tcl
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 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"
```
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.
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.

```tcl
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
```
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:

```tcl
set_module_property STRUCTURAL_COMPOSITION_CALLBACK structural_hierarchy
```

```tcl
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 exported in 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**

```tcl
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.

```tcl
//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
    ...
)
```

Altera Corporation
.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
																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																									}

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
    }
```
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  
  ...  
  ... );
ex...mple_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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>• Updated screen shots Files tab, Qsys Component Editor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Specify Interfaces and Signals in the Qsys Component Editor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Create an HDL File in the Qsys Component Editor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Create an HDL File Using a Template in the Qsys Component Editor.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| 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 is a high-bandwidth structure that allows you to connect IP components to other IP components with various interfaces.

Qsys supports Avalon, AMBA AXI (version 1.0), AMBA AXI4 (version 2.0), AMBA AXI4-Lite (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB (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.

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Specifies the byte address for the lowest byte in the current cycle. There are no restrictions on address alignment.</td>
</tr>
<tr>
<td>Size</td>
<td>Encodes the run-time size of the transaction.</td>
</tr>
<tr>
<td></td>
<td>In conjunction with address, this field describes the segment of the payload that contains valid data for a beat within the packet.</td>
</tr>
<tr>
<td>Address Sideband</td>
<td>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.</td>
</tr>
<tr>
<td>Cache</td>
<td>Carries the AXI cache signals.</td>
</tr>
<tr>
<td>Transaction (Exclusive)</td>
<td>Indicates whether the transaction has exclusive access.</td>
</tr>
<tr>
<td>Transaction (Posted)</td>
<td>Used to indicate non-posted writes (writes that require responses).</td>
</tr>
<tr>
<td>Data</td>
<td>For command packets, carries the data to be written. For read response packets, carries the data that has been read.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Byteenable</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• 1111—Writes full 32 bits</td>
</tr>
<tr>
<td></td>
<td>• 0011—Writes lower 2 bytes</td>
</tr>
<tr>
<td></td>
<td>• 1100—Writes upper 2 bytes</td>
</tr>
<tr>
<td></td>
<td>• 0001—Writes byte 0 only</td>
</tr>
<tr>
<td></td>
<td>• 0010—Writes byte 1 only</td>
</tr>
<tr>
<td></td>
<td>• 0100—Writes byte 2 only</td>
</tr>
<tr>
<td></td>
<td>• 1000—Writes byte 3 only</td>
</tr>
<tr>
<td>Source_ID</td>
<td>The ID of the master or slave that initiated the command or response.</td>
</tr>
<tr>
<td>Destination_ID</td>
<td>The ID of the master or slave to which the command or response is directed.</td>
</tr>
<tr>
<td>Response</td>
<td>Carries the AXI response signals.</td>
</tr>
<tr>
<td>Thread ID</td>
<td>Carries the AXI transaction ID values.</td>
</tr>
<tr>
<td>Byte count</td>
<td>The number of bytes remaining in the transaction, including this beat. Number of bytes requested by the packet.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Burstwrap</td>
<td>The burstwrap value specifies the wrapping behavior of the current burst. The burstwrap value is of the form $2^{m} - 1$. The following types are defined:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• For a burst wrap boundary of size $&lt;m&gt;$, $\text{Burstwrap} = &lt;m&gt; - 1$, or for this case $\text{Burstwrap} = (32 - 1) = 31$ which is $2^5 - 1$.</td>
</tr>
<tr>
<td></td>
<td>• For AXI masters, the burstwrap boundary value ($m$) is based on the different AXBURST:</td>
</tr>
<tr>
<td></td>
<td>• Burstwrap set to all 1’s. For example, for a 6-bit burstwrap, burstwrap is $6'b111111$.</td>
</tr>
<tr>
<td></td>
<td>• For WRAP bursts, burstwrap = AXLEN * size – 1.</td>
</tr>
<tr>
<td></td>
<td>• For FIXED bursts, burstwrap = size – 1.</td>
</tr>
<tr>
<td></td>
<td>• 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 $2^6 - 1$.</td>
</tr>
<tr>
<td>Protection</td>
<td>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.</td>
</tr>
<tr>
<td>QoS</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PKT_TRANS_READ</td>
<td>When asserted, indicates a read transaction.</td>
</tr>
<tr>
<td>1</td>
<td>PKT_TRANS_COMPRESSED_READ</td>
<td>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 byte enables asserted on every cycle.</td>
</tr>
<tr>
<td>2</td>
<td>PKT_TRANS_WRITE</td>
<td>When asserted, indicates a write transaction.</td>
</tr>
<tr>
<td>3</td>
<td>PKT_TRANS_POSTED</td>
<td>When asserted, no response is required.</td>
</tr>
<tr>
<td>4</td>
<td>PKT_TRANS_LOCK</td>
<td>When asserted, indicates arbitration is locked. Applies to write packets.</td>
</tr>
</tbody>
</table>

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.
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 AXQ0 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 finite-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**

<table>
<thead>
<tr>
<th>Connections</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm_master_bfm_0_avalon</td>
<td>Altera UVM Avalon-MM Master BFM</td>
<td>Clock Input, Reset Input, Avalon Memory Mapped Master</td>
</tr>
<tr>
<td>mm_master_bfm_1_axi</td>
<td>Altera AXI3 Master Module</td>
<td>Clock Input, Reset Input, AXI Master</td>
</tr>
<tr>
<td>mm_master_bfm_2_axi</td>
<td>Altera AXI3 Master Module</td>
<td>Clock Input, Reset Input, AXI Master</td>
</tr>
<tr>
<td>mm_slave_bfm_0_avalon</td>
<td>Altera UVM Avalon-MM Slave BFM</td>
<td>Clock Input, Reset Input, Avalon Memory Mapped Slave</td>
</tr>
<tr>
<td>mm_slave_bfm_1_aavalon</td>
<td>Altera UVM Avalon-MM Slave BFM</td>
<td>Clock Input, Reset Input, Avalon Memory Mapped Slave</td>
</tr>
<tr>
<td>mm_slave_bfm_2_axi</td>
<td>Altera AXI3 Slave Module</td>
<td>Clock Input, Reset Input, AXI Slave</td>
</tr>
<tr>
<td>CLOCK_0</td>
<td>Altera Avalon Clock and Reset Source</td>
<td>Clock Output, Reset Output, Avalon Streaming Source, Avalon Streaming Snk</td>
</tr>
</tbody>
</table>

**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.
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.

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.

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 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 backpressures 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**

6. Navigate to the **System Contents** tab.

---

**Send Feedback**
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 priority 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**

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.

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_{\text{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.
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)

<table>
<thead>
<tr>
<th>Burst Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incrementing</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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 <code>INCR9</code> burst is converted into <code>INCR16</code> + <code>INCR2</code> 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.</td>
</tr>
</tbody>
</table>
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 wrap 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 incr burst.

### AXI Narrow-to-Wide Adaptation

<table>
<thead>
<tr>
<th>Burst Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incrementing</td>
<td>The burst (and its response) passes through unmodified. Data and write strobes are placed in the correct output segment.</td>
</tr>
<tr>
<td>Wrapping</td>
<td>The burst (and its response) passes through unmodified.</td>
</tr>
<tr>
<td>Fixed</td>
<td>The burst (and its response) passes through unmodified.</td>
</tr>
</tbody>
</table>

### 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 $f_{\text{max}}$ 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 $f_{\text{max}}$ but higher area. This setting is useful when you have AXI masters or slaves and you want a higher $f_{\text{max}}$.

**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.

<table>
<thead>
<tr>
<th>Burst Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incrementing</td>
<td><strong>Sequential Slave</strong>&lt;br&gt;Bursts that exceed <code>slave_max_burst_length</code> are converted to multiple sequential bursts of a length less than or equal to the <code>slave_max_burst_length</code>. Otherwise, the burst is unconverted. For example, for an Avalon slave with a maximum burst length of 4, an <code>INCR7</code> burst is converted to <code>INCR4 + INCR3</code>.</td>
</tr>
<tr>
<td>Wrapping</td>
<td><strong>Sequential Slave</strong>&lt;br&gt;A WRAP burst is converted to multiple sequential bursts. The sequential bursts are less than or equal to the <code>max_burst_length</code> and respect the transaction’s wrapping boundary. <strong>Wrapping Slave</strong>&lt;br&gt;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.</td>
</tr>
<tr>
<td>Fixed</td>
<td>Fixed bursts are converted to sequential bursts of length 1 that repeatedly access the same address.</td>
</tr>
<tr>
<td>Narrow</td>
<td>All narrow-sized bursts are broken into multiple bursts of length 1.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Burst Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Bursts of length greater than 16 are converted to multiple <code>INCR</code> bursts of a length less than or equal to 16. Bursts of length less than or equal to 16 are not converted.</td>
</tr>
</tbody>
</table>
### Burst Type

| Wrapping | Only Avalon masters with `alwaysBurstMaxBurst = true` are supported. The WRAP burst is passed through if the length is less than or equal to 16. 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 `fMax`. |

### 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:

\[
\text{DECERR} > \text{SLVERR} > \text{OKAY} > \text{EXOKAY}
\]

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

<table>
<thead>
<tr>
<th>Master with Response</th>
<th>Slave with Response</th>
<th>Slave without Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master with Response</td>
<td>Interconnect delivers response from the slave to the master.</td>
<td>Interconnect delivers an <code>OKAY</code> response to the master.</td>
</tr>
<tr>
<td></td>
<td>Response merging or duplication may be necessary for bus sizing.</td>
<td>No need for responses. Master, slave, and interconnect operate without response support.</td>
</tr>
<tr>
<td>Master without Response</td>
<td>Master ignores responses from the slave.</td>
<td></td>
</tr>
</tbody>
</table>

**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 (\(<\text{M}..0\>)\) and the individual slave address widths (\(<\text{S}..0\>) \) and (\(<\text{T}..2\>)\). The address decoding logic also maps only the necessary master address bits to access words in each slave’s address space.
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 $f_{\text{MAX}}$.

**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.
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.
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.

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>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol Width</td>
<td>Width of a single symbol in bits.</td>
</tr>
<tr>
<td>Use Packet</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Avalon-ST Adapter Upstream Source Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Data Width</td>
<td>Controls the data width of the source interface data port.</td>
</tr>
<tr>
<td>Source Top Channel</td>
<td>Maximum number of output channels allowed.</td>
</tr>
<tr>
<td>Source Channel Port Width</td>
<td>Sets the bit width of the source interface channel port. If set to 0, there is no channel port on the sink interface.</td>
</tr>
<tr>
<td>Source Error Port Width</td>
<td>Sets the bit width of the source interface error port. If set to 0, there is no error port on the sink interface.</td>
</tr>
<tr>
<td>Source Error Descriptors</td>
<td>A list of strings that describe the error conditions for each bit of the source interface error signal.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Source Uses Empty Port</td>
<td>Indicates whether the source interface includes the empty port, and whether the sink interface should also include the empty port.</td>
</tr>
<tr>
<td>Source Empty Port Width</td>
<td>Indicates the bit width of the source interface empty port, and sets the bit width of the sink interface empty port.</td>
</tr>
<tr>
<td>Source Uses Valid Port</td>
<td>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.</td>
</tr>
<tr>
<td>Source Uses Ready Port</td>
<td>Indicates whether the sink interface uses the ready port, and if set, configures the source interface to use the ready port.</td>
</tr>
<tr>
<td>Source Ready Latency</td>
<td>Specifies what ready latency to expect from the source interface connected to the adapter's sink interface.</td>
</tr>
<tr>
<td>Sink Data Width</td>
<td>Indicates the bit width of the data port on the sink interface connected to the source interface.</td>
</tr>
<tr>
<td>Sink Top Channel</td>
<td>Maximum number of output channels allowed.</td>
</tr>
<tr>
<td>Sink Channel Port Width</td>
<td>Indicates the bit width of the channel port on the sink interface connected to the source interface.</td>
</tr>
<tr>
<td>Sink Error Port Width</td>
<td>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.</td>
</tr>
<tr>
<td>Sink Error Descriptors</td>
<td>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.</td>
</tr>
<tr>
<td>Sink Uses Empty Port</td>
<td>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.</td>
</tr>
<tr>
<td>Sink Empty Port Width</td>
<td>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.</td>
</tr>
<tr>
<td>Sink Uses Valid Port</td>
<td>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.</td>
</tr>
<tr>
<td>Sink Uses Ready Port</td>
<td>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.</td>
</tr>
</tbody>
</table>
Channel Adapter

The channel adapter provides adaptations between interfaces that have different channel signal widths.

Table 6-10: Channel Adapter Adaptations

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description of Adapter Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>The source uses channels, but the sink does not.</td>
<td>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.</td>
</tr>
<tr>
<td>The sink has channel, but the source does not.</td>
<td>Qsys gives a warning at generation time, and the channel inputs to the sink are all tied to a logical 0.</td>
</tr>
<tr>
<td>The source and sink both support channels, and the source's maximum channel number is less than the sink's maximum channel number.</td>
<td>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.</td>
</tr>
<tr>
<td>The source and sink both support channels, but the source's maximum channel number is greater than the sink's maximum channel number.</td>
<td>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.</td>
</tr>
</tbody>
</table>

Avalon-ST Channel Adapter Input Interface Parameters

Table 6-11: Avalon-ST Channel Adapter Input Interface Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Signal Width (bits)</td>
<td>Width of the input channel signal in bits</td>
</tr>
<tr>
<td>Max Channel</td>
<td>Maximum number of input channels allowed.</td>
</tr>
</tbody>
</table>
### Avalon-ST Channel Adapter Output Interface Parameters

**Table 6-12: Avalon-ST Channel Adapter Output Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Signal Width (bits)</td>
<td>Width of the output channel signal in bits.</td>
</tr>
<tr>
<td>Max Channel</td>
<td>Maximum number of output channels allowed.</td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Bits Per Symbol</td>
<td>Number of bits for each symbol in a transfer.</td>
</tr>
<tr>
<td>Include Packet Support</td>
<td>When the Avalon-ST Channel adapter supports packets, the startofpacket, endofpacket, and optional empty signals are included on its sink and source interfaces.</td>
</tr>
<tr>
<td>Include Empty Signal</td>
<td>Indicates whether an empty signal is required.</td>
</tr>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per transfer.</td>
</tr>
<tr>
<td>Support Backpressure with the ready signal</td>
<td>Indicates whether a ready signal is required.</td>
</tr>
<tr>
<td>Ready Latency</td>
<td>Specifies the ready latency to expect from the sink connected to the module's source interface.</td>
</tr>
<tr>
<td>Error Signal Width (bits)</td>
<td>Bit width of the error signal.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>A list of strings that describes what each bit of the error signal represents.</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Condition</th>
<th>Description of Adapter Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>The source and sink’s bits per symbol parameters are different.</td>
<td>The connection cannot be made.</td>
</tr>
</tbody>
</table>
| 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**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per transfer.</td>
</tr>
<tr>
<td>Include Empty Signal</td>
<td>Indicates whether an empty signal is required.</td>
</tr>
</tbody>
</table>

**Avalon-ST Data Format Adapter Output Interface Parameters**

**Table 6-16: Avalon-ST Data Format Adapter Output Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per transfer.</td>
</tr>
<tr>
<td>Include Empty Signals</td>
<td>Indicates whether an empty signal is required.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Bits Per Symbol</td>
<td>Number of bits for each symbol in a transfer.</td>
</tr>
<tr>
<td>Include Packet Support</td>
<td>When the Avalon-ST Data Format adapter supports packets, Qsys uses \texttt{startofpacket}, \texttt{endofpacket}, and \texttt{empty} signals.</td>
</tr>
<tr>
<td>Channel Signal Width (bits)</td>
<td>Width of the output channel signal in bits.</td>
</tr>
<tr>
<td>Max Channel</td>
<td>Maximum number of channels allowed.</td>
</tr>
<tr>
<td>Read Latency</td>
<td>Specifies the ready latency to expect from the sink connected to the module's source interface.</td>
</tr>
<tr>
<td>Error Signal Width (bits)</td>
<td>Width of the error signal output in bits.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>A list of strings that describes what each bit of the error signal represents.</td>
</tr>
</tbody>
</table>

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 \texttt{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

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Signal Width (bits)</td>
<td>The width of the error signal. Valid values are 0–256 bits. Type 0 if the error signal is not used.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Avalon-ST Error Adapter Output Interface Parameters

**Table 6-19: Avalon-ST Error Adapter Output Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Signal Width (bits)</td>
<td>The width of the error signal. Valid values are 0–256 bits. Type 0 if you do not need to send error values.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support Backpressure with the ready signal</td>
<td>Turn on this option to add the backpressure functionality to the interface.</td>
</tr>
<tr>
<td>Ready Latency</td>
<td>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.</td>
</tr>
<tr>
<td>Channel Signal Width (bits)</td>
<td>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.</td>
</tr>
<tr>
<td>Max Channel</td>
<td>The maximum number of channels that the interface supports. Valid values are 0–255.</td>
</tr>
<tr>
<td>Data Bits Per Symbol</td>
<td>Number of bits per symbol.</td>
</tr>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per active transfer.</td>
</tr>
<tr>
<td>Include Packet Support</td>
<td>Turn on this option if the connected interfaces support a packet protocol, including the startof-packet, endofpacket and empty signals.</td>
</tr>
<tr>
<td>Include Empty Signal</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 \texttt{ready} signal, and those that do not. The timing adapter treats all signals other than the \texttt{ready} and \texttt{valid} signals as payload, and simply drives them from the source to the sink.

Table 6-21: Timing Adapter Adaptations

<table>
<thead>
<tr>
<th>Condition</th>
<th>Adaptation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The source has \texttt{ready}, but the sink does not.</td>
<td>In this case, the source can respond to \texttt{backpressure}, but the sink never needs to apply it.</td>
</tr>
<tr>
<td></td>
<td>The \texttt{ready} input to the source interface is connected directly to logical 1.</td>
</tr>
<tr>
<td>The source does not have \texttt{ready}, but the sink does.</td>
<td>The sink may apply \texttt{backpressure}, but the source is unable to respond to it. There is no logic</td>
</tr>
<tr>
<td></td>
<td>that the adapter can insert that prevents data loss when the source asserts \texttt{valid} but the</td>
</tr>
<tr>
<td></td>
<td>sink is not ready. The adapter provides simulation time error messages if data is lost. The user is</td>
</tr>
<tr>
<td></td>
<td>presented with a warning, and the connection is allowed.</td>
</tr>
<tr>
<td>The source and sink both support backpressure, but the sink's ready latency</td>
<td>The source responds to \texttt{ready} assertion or deassertion faster than the sink requires it. A</td>
</tr>
<tr>
<td>is greater than the source’s.</td>
<td>number of pipeline stages equal to the difference in ready latency are inserted in the \texttt{ready}</td>
</tr>
<tr>
<td></td>
<td>path from the sink back to the source, causing the source and the sink to see the same cycles as</td>
</tr>
<tr>
<td></td>
<td>\texttt{ready} cycles.</td>
</tr>
<tr>
<td>The source and sink both support backpressure, but the sink's ready latency</td>
<td>The source cannot respond to \texttt{ready} assertion or deassertion in time to satisfy the sink. A</td>
</tr>
<tr>
<td>is less than the source’s.</td>
<td>FIFO whose depth is equal to the difference in ready latency is inserted to compensate for the source’s</td>
</tr>
<tr>
<td></td>
<td>inability to respond in time.</td>
</tr>
</tbody>
</table>

Avalon-ST Timing Adapter Input Interface Parameters

Table 6-22: Avalon-ST Timing Adapter Input Interface Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support Backpressure with the \texttt{ready} signal</td>
<td>Indicates whether a \texttt{ready} signal is required.</td>
</tr>
<tr>
<td>Read Latency</td>
<td>Specifies the ready latency to expect from the sink</td>
</tr>
<tr>
<td></td>
<td>connected to the module’s source interface.</td>
</tr>
</tbody>
</table>
### Avalon-ST Timing Adapter Output Interface Parameters

**Table 6-23: Avalon-ST Timing Adapter Output Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include Valid Signal</td>
<td>Indicates whether the sink interface requires a valid signal.</td>
</tr>
<tr>
<td>Support Backpressure with the ready signal</td>
<td>Indicates whether a <code>ready</code> signal is required.</td>
</tr>
<tr>
<td>Read Latency</td>
<td>Specifies the ready latency to expect from the sink connected to the module’s source interface.</td>
</tr>
<tr>
<td>Include Valid Signal</td>
<td>Indicates whether the sink interface requires a valid signal.</td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Bits Per Symbol</td>
<td>Number of bits for each symbol in a transfer.</td>
</tr>
<tr>
<td>Include Packet Support</td>
<td>Turn this option on if the connected interfaces support a packet protocol, including the <code>startofpacket</code>, <code>endofpacket</code> and <code>empty</code> signals.</td>
</tr>
<tr>
<td>Include Empty Signal</td>
<td>Turn this option on if the cycle that includes the <code>endofpacket</code> signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.</td>
</tr>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per active transfer.</td>
</tr>
<tr>
<td>Channel Signal Width (bits)</td>
<td>Width of the output channel signal in bits.</td>
</tr>
<tr>
<td>Max Channel</td>
<td>Maximum number of output channels allowed.</td>
</tr>
<tr>
<td>Error Signal Width (bits)</td>
<td>Width of the output <code>error</code> signal in bits.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>A list of strings that describes errors.</td>
</tr>
</tbody>
</table>
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 $\text{irq}[31:0]$ to the receiver, and maps slave IRQ signals to the bits of $\text{irq}[31:0]$. Any unassigned bits of $\text{irq}[31:0]$ 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 receiver 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.

**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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Output</td>
<td>1 bit</td>
<td>A single bit wide port role, which provides synchronization for internal logic.</td>
</tr>
</tbody>
</table>
Table 6-26: HSSI Serial Clock Source Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Derived</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockRate</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>The frequency of the clock driven byte HSSI Serial Clock Source interface.</td>
</tr>
</tbody>
</table>

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:

```tcl
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Output</td>
<td>1</td>
<td>A single bit wide port role, which provides synchronization for internal logic</td>
</tr>
</tbody>
</table>

Table 6-28: HSSI Serial Clock Sink Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Derived</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockRate</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>The frequency of the clock driven by the HSSI Serial Clock Source interface. When you specify a <strong>clockRate</strong> greater than 0, then this interface can be driven only at that rate.</td>
</tr>
</tbody>
</table>

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.

```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.

```tcl
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 **serializationFactor**. **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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Output</td>
<td>1 to 24 bits</td>
<td>A multiple bit wide port role which provides synchronization for internal logic.</td>
</tr>
</tbody>
</table>

**Table 6-30: HSSI Bonded Clock Source Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Derived</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockRate</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>The frequency of the clock driven byte HSSI Serial Clock Source interface.</td>
</tr>
<tr>
<td>serialization</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 **serializationFactor**. **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 Sink is valid and does not generate error messages.
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Output</td>
<td>1 to 24 bits</td>
<td>A multiple bit wide port role which provides synchronization for internal logic.</td>
</tr>
</tbody>
</table>

### Table 6-32: HSSI Bonded Clock Source Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Derived</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockRate</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>The frequency of the clock driven by the HSSI Serial Clock Source interface.</td>
</tr>
<tr>
<td>serialization</td>
<td>long</td>
<td>0</td>
<td>No</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 `serializationFactor` 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.

```tcl
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 enable 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 enable 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: HSSI Bonded Clock Instantiated in a Composed Component

```tcl
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 reset request 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 of 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.
### Reset Sequencer Parameters

#### Table 6-33: Reset Sequencer Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of reset outputs</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Number of reset inputs</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Minimum reset assertion time</strong></td>
<td>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.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Enable Reset Sequencer CSR</td>
<td>Enables CSR functionality of the Reset Sequencer through an Avalon interface.</td>
</tr>
<tr>
<td>reset_out#</td>
<td>Lists the reset output signals. Set the parameters in the other columns for each reset signal in the table.</td>
</tr>
<tr>
<td>ASRT Seq#</td>
<td>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.</td>
</tr>
<tr>
<td>ASRT Cycle#</td>
<td>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 to 1023.</td>
</tr>
<tr>
<td>DSRT Seq#</td>
<td>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.</td>
</tr>
<tr>
<td>DSRT Cycle#/Deglitch#</td>
<td>Number of cycles to wait before de-asserting or de-glitching the reset. If the USE_DSRT_QUAL parameter is set to 0, specifies the number of cycles to wait before de-asserting the reset. If USE_DSRT_QUAL is set to 1, specifies the number of cycles to de-glitch the input reset_dsrt_qual signal. This value determines either the DSRT_DELAY, or the DSRT_QALCNT value in the component HDL, depending on the USE_DSRT_QUAL parameter setting. The range is 0 to 1023.</td>
</tr>
<tr>
<td>USE_DSRT_QUAL</td>
<td>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.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RO</td>
<td>0</td>
<td>Reset Active—Indicates that the sequencer is currently active in reset sequence (assertion or de-assertion).</td>
</tr>
<tr>
<td>30</td>
<td>RW1C</td>
<td>0</td>
<td><strong>Reset Asserted and waiting for SW to proceed:</strong>—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.</td>
</tr>
<tr>
<td>29</td>
<td>RW1C</td>
<td>0</td>
<td><strong>Reset De-asserted and waiting for SW to proceed:</strong>—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.</td>
</tr>
<tr>
<td>28:26</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>25:16</td>
<td>RW1C</td>
<td>0</td>
<td><strong>Reset de-assertion input qualification signal reset_dsrt_qual [9:0] status</strong>—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.</td>
</tr>
<tr>
<td>15:12</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>11</td>
<td>RW1C</td>
<td>0</td>
<td><strong>reset_in9 was triggered</strong>—Indicates that reset_in9 triggered the reset. Cleared by software by writing 1 to this bit location.</td>
</tr>
</tbody>
</table>
### Table 6-35: Values for the Interrupt Enable Register at Offset 0x04

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Bit</td>
<td>Attribute</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>30</td>
<td>RW</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>29</td>
<td>RW</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>28:26</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>25:16</td>
<td>RW</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>15:12</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>11</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in9 Enable—When set, the IRQ is set when the reset_in9 trigger status bit is set.</td>
</tr>
<tr>
<td>10</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in8 Enable—When set, the IRQ is set when the reset_in8 trigger status bit is set.</td>
</tr>
<tr>
<td>9</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in7 Enable—When set, the IRQ is set when the reset_in7 trigger status bit is set.</td>
</tr>
<tr>
<td>8</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in6 Enable—When set, the IRQ is set when the reset_in6 trigger status bit is set.</td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in5 Enable—When set, the IRQ is set when the reset_in5 trigger status bit is set.</td>
</tr>
<tr>
<td>6</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in4 Enable—When set, the IRQ is set when the reset_in4 trigger status bit is set.</td>
</tr>
<tr>
<td>5</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in3 Enable—When set, the IRQ is set when the reset_in3 trigger status bit is set.</td>
</tr>
<tr>
<td>4</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in2 Enable—When set, the IRQ is set when the reset_in2 trigger status bit is set.</td>
</tr>
<tr>
<td>3</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in1 Enable—When set, the IRQ is set when the reset_in1 trigger status bit is set.</td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on reset_in0 Enable—When set, the IRQ is set when the reset_in0 trigger status bit is set.</td>
</tr>
<tr>
<td>Bit</td>
<td>Attribute</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>RW</td>
<td>0</td>
<td><strong>Interrupt on Software triggered reset Enable</strong>—When set, the IRQ is set when the software triggered reset status bit is set.</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>0</td>
<td><strong>Interrupt on Power-On-Reset Enable</strong>—When set, the IRQ is set when the power-on-reset status bit is set.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:3</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td>0</td>
<td><strong>Enable SW sequenced reset entry</strong>—Enable a software sequenced reset entry sequence. Timer delays and input qualification are ignored, and only the software can sequence the entry.</td>
</tr>
<tr>
<td>1</td>
<td>RW</td>
<td>0</td>
<td><strong>Enable SW sequenced reset bring up</strong>—Enable a software sequenced reset bring up sequence. Timer delays and input qualification are ignored, and only the software can sequence the bring up.</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0</td>
<td><strong>Initiate Reset Sequence</strong>—Reset Sequencer writes this bit to 1 a single time in order to trigger the hardware sequenced warm reset. Reset Sequencer verifies that <strong>Reset Active</strong> is 0 before setting this bit, and always reads the value 0. To monitor this sequence, verify that <strong>Reset Active</strong> is asserted, and then subsequently de-asserted.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:10</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
### Table 6-38: Values for the Reset Sequencer Software Sequenced Bring Up Control Register at Offset 0x10

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:10</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>9:0</td>
<td>RW</td>
<td>3FF</td>
<td>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 the 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).</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>25:16</td>
<td>WO</td>
<td>0</td>
<td>Reset Overwrite Trigger Enable — This is a per-bit control trigger bit for the overwrite value to take effect.</td>
</tr>
<tr>
<td>15:10</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>9:0</td>
<td>WO</td>
<td>0</td>
<td>reset_outN Reset Overwrite Value — This is a per-bit control of the reset_outN bit. The Reset Sequencer can use this to forcefully drive the reset to a specific value. A value of 1 sets the reset_outN. A value of 0 clears the reset_outN. A write to this register only takes effect if the corresponding trigger bit in this register is set.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:10</td>
<td>RO</td>
<td>0</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
| 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 Register 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 Register. When set, it indicates that Reset Sequencer has completed initiating a reset through the sequencer.

Software clears bit 1 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**

1. Software sets the Enable software-sequenced reset entry bit (bit 2 of the Control Register).
2. Software sets up which reset sequence it wants to control (or all reset outputs) with the Pre-reset-software-sequenced reset entry enable bit.
3. Software enables Interrupt on reset asserted so that the Reset Sequencer waits for software upon setting the IRQ in the sequence.
4. Setup is complete.
5. Software asserts reset.
6. Hardware sequences a reset where the software has previously set up the Reset Sequencer to wait for a software signal.
7. Reset Sequencer asserts an IRQ.
8. Software acknowledges that the reset is asserted and bit 30 of the Status Register is set.
9. Software clears Reset asserted and waiting for software to proceed bit 30 of the Status Register and the Reset Sequencer proceeds with the sequence.
10. 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 $f_{\text{MAX}}$ 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.

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 pipeline 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 Equal15-0 combout
   3.346 0.100 mm_domain_0 addr_router_001 Equal5~1 dataa
   3.685 0.339 mm_domain_0 addr_router_001 Equal15-1 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 memory elements as a DECRError 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, \texttt{AWSIZE} or \texttt{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 \texttt{0x01}, and a burstcount of 2 to a 32-bit Avalon slave, the starting address is: \texttt{0x00}.
- For a 32-bit AXI master that issues a read command with an unaligned address starting at address \texttt{0x01}, with 4-bytes to an 8-bit AXI slave, the starting address is: \texttt{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 \texttt{PSELx} in Qsys. As a workaround, you can group \texttt{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, \texttt{RRESP/BRESP=SLVERR}. For the case when the slave does not use SLVERR signal, an \texttt{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 \texttt{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 \texttt{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 data width 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 \texttt{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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>associatedClock</td>
<td>string</td>
<td>Name of associated clock interface.</td>
</tr>
<tr>
<td>associatedReset</td>
<td>string</td>
<td>Name of associated reset interface</td>
</tr>
</tbody>
</table>
### AXI4 Streaming Connection Point Signals

#### Table 6-42: AXI4 Stream Connection Point Signals

<table>
<thead>
<tr>
<th>Port Role</th>
<th>Width</th>
<th>Master Direction</th>
<th>Slave Direction</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>tvalid</td>
<td>1</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>tready</td>
<td>1</td>
<td>Input</td>
<td>Output</td>
<td>No</td>
</tr>
<tr>
<td>tdata(4)</td>
<td>8:4096</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tstrb</td>
<td>1:512</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tkeep</td>
<td>1:512</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tid(5)</td>
<td>1:8</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tdest(6)</td>
<td>1:4</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tuser(7)</td>
<td>1:4096</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tlast</td>
<td>1</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
</tbody>
</table>

#### 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 multiple 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

<table>
<thead>
<tr>
<th>Global</th>
<th>Write Address Channel</th>
<th>Write Data Channel</th>
<th>Write Response Channel</th>
<th>Read Address Channel</th>
<th>Read Data Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLK</td>
<td>AWVALID</td>
<td>WVALID</td>
<td>BVALID</td>
<td>ARVALID</td>
<td>RVALID</td>
</tr>
<tr>
<td>ARESETn</td>
<td>AWREADY</td>
<td>WREADY</td>
<td>BREADY</td>
<td>ARREADY</td>
<td>RREADY</td>
</tr>
<tr>
<td>-</td>
<td>AWADDR</td>
<td>WDATA</td>
<td>BRESP</td>
<td>ARADDR</td>
<td>RDATA</td>
</tr>
<tr>
<td>-</td>
<td>AWPROT</td>
<td>WSTRB</td>
<td>-</td>
<td>ARPROT</td>
<td>RRESP</td>
</tr>
</tbody>
</table>

### 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 interface 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>araddr</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arburst</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>arcache</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>arid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>arlen</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>arlock</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>arprot</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>arready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>arsize</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>aruser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>awaddr</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awburst</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>awcache</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>awid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>awlen</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>awlock</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>awprot</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>awready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>awsize</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>awuser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>bid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>bready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>Direction</td>
<td>Width</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>bresp</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>bvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>rdata</td>
<td>input</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>rid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>rlast</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>rready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rresp</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>rvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wdata</td>
<td>output</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>wid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>wlast</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wstrb</td>
<td>output</td>
<td>1, 2, 4, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>wvalid</td>
<td>output</td>
<td>1</td>
</tr>
</tbody>
</table>

**AXI Slave Interface Signal Types**

Table 6-45: AXI Slave Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>araddr</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arburst</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>arcache</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>arid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>arlen</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>arlock</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>arprot</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>arready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>arsize</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>aruser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>awaddr</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awburst</td>
<td>input</td>
<td>2</td>
</tr>
</tbody>
</table>
### AXI4 Master Interface Signal Types

#### Table 6-46: AXI4 Master Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>awcache</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>awid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>awlen</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>awlock</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>awprot</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>awready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>awsize</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>awuser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>bid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>bready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>bresp</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>bvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rdata</td>
<td>output</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>rid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>rlast</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>rresp</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>rvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wdata</td>
<td>input</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>wid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>wlast</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wstrb</td>
<td>input</td>
<td>1, 2, 4, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>wvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>Direction</td>
<td>Width</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>--------</td>
</tr>
<tr>
<td>arcache</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>arid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>arlen</td>
<td>output</td>
<td>8</td>
</tr>
<tr>
<td>arlock</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>arprot</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>arready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>arregion</td>
<td>output</td>
<td>1 - 4</td>
</tr>
<tr>
<td>arsize</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>aruser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>awaddr</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awburst</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>awcache</td>
<td>output</td>
<td>4</td>
</tr>
<tr>
<td>awid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>awlen</td>
<td>output</td>
<td>8</td>
</tr>
<tr>
<td>awlock</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>awprot</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>awqos</td>
<td>output</td>
<td>1 - 4</td>
</tr>
<tr>
<td>awready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>awregion</td>
<td>output</td>
<td>1 - 4</td>
</tr>
<tr>
<td>awsize</td>
<td>output</td>
<td>3</td>
</tr>
<tr>
<td>awuser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>bid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>bready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>bresp</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>buser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>bvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>rdata</td>
<td>input</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>rid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>rlast</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>Direction</td>
<td>Width</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>rready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>resp</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>ruser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>rvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wdata</td>
<td>output</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>wid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>wlast</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wstrb</td>
<td>output</td>
<td>1, 2, 4, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>wuser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>wvalid</td>
<td>output</td>
<td>1</td>
</tr>
</tbody>
</table>

**AXI4 Slave Interface Signal Types**

Table 6-47: AXI4 Slave Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>araddr</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arburst</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>arcache</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>arid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>arlen</td>
<td>input</td>
<td>8</td>
</tr>
<tr>
<td>arlock</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>arprot</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>arqos</td>
<td>input</td>
<td>1 - 4</td>
</tr>
<tr>
<td>arready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>arregion</td>
<td>input</td>
<td>1 - 4</td>
</tr>
<tr>
<td>arsize</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>aruser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>arvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>awaddr</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awburst</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>awcache</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>Name</td>
<td>Direction</td>
<td>Width</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>awid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
<tr>
<td>awlen</td>
<td>input</td>
<td>8</td>
</tr>
<tr>
<td>awlock</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>awprot</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>awqos</td>
<td>input</td>
<td>1 - 4</td>
</tr>
<tr>
<td>awready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>awregion</td>
<td>inout</td>
<td>1 - 4</td>
</tr>
<tr>
<td>awsize</td>
<td>input</td>
<td>3</td>
</tr>
<tr>
<td>awuser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>awvalid</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>bid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>bready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>bresp</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>bvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rdata</td>
<td>output</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>rid</td>
<td>output</td>
<td>1 - 18</td>
</tr>
<tr>
<td>rlast</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rready</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>rresp</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>ruser</td>
<td>output</td>
<td>1 - 64</td>
</tr>
<tr>
<td>rvalid</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wdata</td>
<td>input</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>wlast</td>
<td>input</td>
<td>1</td>
</tr>
<tr>
<td>wready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>wstrb</td>
<td>input</td>
<td>1, 2, 4, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>wuser</td>
<td>input</td>
<td>1 - 64</td>
</tr>
<tr>
<td>wvalid</td>
<td>input</td>
<td>1</td>
</tr>
</tbody>
</table>
AXI4 Stream Master and Slave Interface Signal Types

Table 6-48: AXI4 Stream Master and Slave Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Master Direction</th>
<th>Slave Direction</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>tvalid</td>
<td>1</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>tready</td>
<td>1</td>
<td>Input</td>
<td>Output</td>
<td>No</td>
</tr>
<tr>
<td>tdata</td>
<td>8:4096</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tstrb</td>
<td>1:512</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tkeep</td>
<td>1:512</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tid</td>
<td>1:8</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tdest</td>
<td>1:4</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tuser</td>
<td>1</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tlast</td>
<td>1:4096</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
</tbody>
</table>

APB Interface Signal Types

Table 6-49: APB Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Direction APB Master</th>
<th>Direction APB Slave</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>paddr</td>
<td>[1:32]</td>
<td>output</td>
<td>input</td>
<td>yes</td>
</tr>
<tr>
<td>psel</td>
<td>[1:16]</td>
<td>output</td>
<td>input</td>
<td>yes</td>
</tr>
<tr>
<td>penable</td>
<td>1</td>
<td>output</td>
<td>input</td>
<td>yes</td>
</tr>
<tr>
<td>pwrite</td>
<td>1</td>
<td>output</td>
<td>input</td>
<td>yes</td>
</tr>
<tr>
<td>pwdata</td>
<td>[1:32]</td>
<td>output</td>
<td>input</td>
<td>yes</td>
</tr>
<tr>
<td>prdata</td>
<td>[1:32]</td>
<td>input</td>
<td>output</td>
<td>yes</td>
</tr>
<tr>
<td>pslverr</td>
<td>1</td>
<td>input</td>
<td>output</td>
<td>no</td>
</tr>
<tr>
<td>pready</td>
<td>1</td>
<td>input</td>
<td>output</td>
<td>yes</td>
</tr>
<tr>
<td>paddr31</td>
<td>1</td>
<td>output</td>
<td>input</td>
<td>no</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental Signals</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>1 - 64</td>
<td>Master → Slave</td>
<td>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.</td>
</tr>
<tr>
<td>byteenable</td>
<td>2, 4, 8, 16, 32, 64, 128</td>
<td>Master → Slave</td>
<td>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 &lt;n&gt; of byteenable indicates whether byte &lt;n&gt; 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</td>
</tr>
<tr>
<td>Signal Role</td>
<td>Width</td>
<td>Direction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>debugaccess</td>
<td>1</td>
<td>Master → Slave</td>
<td>When asserted, allows the Nios II processor to write on-chip memories configured as ROMs.</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>Master → Slave</td>
<td>Asserted to indicate a read transfer. If present, readdata is required.</td>
</tr>
<tr>
<td>read_n</td>
<td></td>
<td>Master → Slave</td>
<td></td>
</tr>
<tr>
<td>readdata</td>
<td>8,16, 32, 64,128, 256, 512, 1024</td>
<td>Slave → Master</td>
<td>The readdata driven from the slave to the master in response to a read transfer.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>response [1:0]</td>
<td>2</td>
<td>Slave → Master</td>
<td>The response signal is an optional signal that carries the response status. <strong>Note:</strong> Because the signal is shared, an interface cannot issue or accept a write response and a read response in the same clock cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 00: OKAY—Successful response for a transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 01: RESERVED—Encoding is reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 10: SLAVEERROR—Error from an endpoint slave. Indicates an unsuccessful transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 11: DECODEERROR—Indicates attempted access to an undefined location.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>1</td>
<td>Master → Slave</td>
<td>Asserted to indicate a write transfer. If present, writedata is required.</td>
</tr>
<tr>
<td>write_n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>writedata</td>
<td>8,16, 32, 64, 128, 256, 512, 1024</td>
<td>Master → Slave</td>
<td>Data for write transfers. The width must be the same as the width of readdata if both are present.</td>
</tr>
</tbody>
</table>

**Wait-State Signals**

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>1</td>
<td>Master → Slave</td>
<td>lock ensures that once a master wins arbitration, it maintains access to the slave for</td>
</tr>
</tbody>
</table>
### Signal Role

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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. |
<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readdatavalid</td>
<td>1</td>
<td>Slave → Master</td>
<td>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 <em>Pipelined Read Transfer with Variable Latency</em>. 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.</td>
</tr>
<tr>
<td>readdatavalid_n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>writeresponsevalid</td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>burstcount</td>
<td>1–11</td>
<td>Master → Slave</td>
<td>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 &lt;n&gt; can encode a max burst of size 2(&lt;n&gt;−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: &lt;address_w&gt; ≥ &lt;burstcount_w&gt; +\log_2(&lt;symbols_per_word_of_interface&gt;). For bursting masters and slaves using word addresses, the \log_2 term above is omitted.</td>
</tr>
</tbody>
</table>
### Signal Role

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginburst-transfer</td>
<td>1</td>
<td>Interconnect → Slave</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>1 – 128</td>
<td>Source → Sink</td>
<td>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.</td>
</tr>
<tr>
<td>data</td>
<td>1 – 4,096</td>
<td>Source → Sink</td>
<td>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.</td>
</tr>
<tr>
<td>error</td>
<td>1 – 256</td>
<td>Source → Sink</td>
<td>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 errorDescriptor property.</td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>Output</td>
<td>Yes</td>
<td>An output clock signal.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>Input</td>
<td>Yes</td>
<td>A clock signal. Provides synchronization for internal logic and for other interfaces.</td>
</tr>
</tbody>
</table>
Avalon Conduit Signal Roles

Table 6-54: Conduit Signal Roles

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;any&gt;</td>
<td>&lt;n&gt;</td>
<td>In, out, or</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bidirectional</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>1</td>
<td>Master → Slave</td>
<td>Yes</td>
<td>The meaning of request depends on the state of the grant signal, as the following rules dictate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When request is asserted and grant is deasserted, request is requesting access for the current cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>grant</td>
<td>1</td>
<td>Slave → Master</td>
<td>Yes</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;name&gt;_in</td>
<td>1 – 1024</td>
<td>Slave → Master</td>
<td>No</td>
<td>The input signal of a logical tristate signal.</td>
</tr>
<tr>
<td>Signal Role</td>
<td>Width</td>
<td>Direction</td>
<td>Required</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>&lt;name&gt;_out</td>
<td>1 – 1024</td>
<td>Master → Slave</td>
<td>No</td>
<td>The output signal of a logical tristate signal.</td>
</tr>
<tr>
<td>&lt;name&gt;_outen</td>
<td>1</td>
<td>Master → Slave</td>
<td>No</td>
<td>The output enable for a logical tristate signal.</td>
</tr>
</tbody>
</table>
# Avalon Tri-State Slave Interface Signal Types

## Table 6-56: Tri-state Slave Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>1 - 32</td>
<td>input</td>
<td>No</td>
<td>Address lines to the slave port. Specifies a byte offset into the slave’s address space.</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>input</td>
<td>No</td>
<td>Read-request signal. Not required if the slave port never outputs data. If present, data must also be used.</td>
</tr>
<tr>
<td>read_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>1</td>
<td>input</td>
<td>No</td>
<td>Write-request signal. Not required if the slave port never receives data from a master. If present, data must also be present, and writebyteenable cannot be present.</td>
</tr>
<tr>
<td>write_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chipselect</td>
<td>1</td>
<td>input</td>
<td>No</td>
<td>When present, the slave port ignores all Avalon-MM signals unless chipselect is asserted. chipselect is always present in combination with read or write</td>
</tr>
<tr>
<td>chipselect_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outputenable</td>
<td>1</td>
<td>input</td>
<td>Yes</td>
<td>Output-enable signal. When deasserted, a tri-state slave port must not drive its data lines otherwise data contention may occur.</td>
</tr>
<tr>
<td>outputenable_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
<td>bidir</td>
<td>No</td>
<td>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.</td>
</tr>
<tr>
<td>Name</td>
<td>Width</td>
<td>Direction</td>
<td>Required</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
<td>-----------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>byteenable</td>
<td>2, 4, 8, 16, 32, 64, 128</td>
<td>input</td>
<td>No</td>
<td>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</td>
</tr>
<tr>
<td>byteenable_n</td>
<td>2, 4, 8, 16, 32, 64, 128</td>
<td>input</td>
<td>No</td>
<td>Equivalent to the logical AND of the byteenable and write signals. When used, the write signal is not used.</td>
</tr>
<tr>
<td>begintransfer1</td>
<td>1</td>
<td>input</td>
<td>No</td>
<td>Asserted for the first cycle of each transfer.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq</td>
<td>1</td>
<td>Output</td>
<td>Yes</td>
<td>Interrupt Request. A slave asserts irq when it needs service. The interrupt receiver determines the relative priority of the interrupts.</td>
</tr>
<tr>
<td>irq_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Avalon Interrupt Receiver Signal Roles

Table 6-58: Interrupt Receiver Signal Roles

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq</td>
<td>1–32</td>
<td>Input</td>
<td>Yes</td>
<td>irq is an &lt;n&gt;-bit vector, where each bit corresponds directly to one IRQ sender with no inherent assumption of priority.</td>
</tr>
</tbody>
</table>

Document Revision History

The table below indicates edits made to the Qsys Interconnect content since its creation.

Table 6-59: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>• Fixed Priority Arbitration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Read and Write Responses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Error Correction Coding (ECC) in Qsys Interconnect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: response [1:0], Avalon Memory-Mapped Interface Signal Roles.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added writeresponsevalid, Avalon Memory-Mapped Interface Signal Roles.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• Read error responses, Avalon Memory-Mapped Interface Signal, response.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Burst Adapter Implementation Options: Generic converter (slower, lower area), Per-burst-type converter (faster, higher area).</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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.
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.
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.
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).
You can also design systems that process multiple data channels by instantiating the same subsystem for each channel. This approach is easier to maintain than a larger, non-hierarchical system. Additionally, such systems are easier to scale because you can calculate the required resources as a multiple of the subsystem requirements.

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 categorize 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.
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.
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.
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
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.

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
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 $f_{\text{MAX}}$.

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 $f_{\text{MAX}}$ 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 $f_{\text{MAX}}$. 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 $f_{\text{MAX}}$ 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.
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 $f_{\text{MAX}}$ 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.
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 $f_{\text{MAX}}$ 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 $f_{\text{MAX}}$ of the system without sacrificing concurrency.
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.
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.
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

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 get 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 \texttt{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 \texttt{burstcounts} equal to the amount of data that is ready. For example, if you create a custom bursting write master with a maximum \texttt{burstcount} of eight, but only three words of data are ready, you can present a \texttt{burstcount} of three. This strategy does not result in optimal use of the system bandwidth 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 \texttt{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 \texttt{burstcount} to the number of sequential addresses, and then reset the \texttt{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 \texttt{burstcount} paths between the master and slave interfaces.

### 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**

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
<th>Pipeline Management Logic Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>No pipeline</td>
<td>No Pipeline</td>
<td>Qsys interconnect does not instantiate logic to handle pipeline latency.</td>
</tr>
<tr>
<td>No pipeline</td>
<td>Pipelined with fixed or variable latency</td>
<td>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.</td>
</tr>
<tr>
<td>Pipelined</td>
<td>No pipeline</td>
<td>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.</td>
</tr>
<tr>
<td>Pipelined</td>
<td>Pipelined with fixed latency</td>
<td>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.</td>
</tr>
<tr>
<td>Pipelined</td>
<td>Pipelined with variable latency</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 $f_{\text{MAX}}$ 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.
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.

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.
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.
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.
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.
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 polarities as needed.
Figure 7-27: Reset Signal (Active-High)
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.
You can combine multiple reset sources to reset a particular component.

**Figure 7-30: Combine Multiple Reset Sources**

<table>
<thead>
<tr>
<th>Use</th>
<th>Connections</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>clk_0</strong></td>
<td>Clock Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk_in</td>
<td>Clock Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk_in_reset</td>
<td>Reset Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk</td>
<td>Clock Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk_reset</td>
<td>Reset Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>reset_bridge_0</strong></td>
<td>Reset Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in_reset</td>
<td>Reset Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>out_reset</td>
<td>Reset Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>mm_bridge_0</strong></td>
<td>Avalon-MM Pipeline Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk</td>
<td>Clock Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s0</td>
<td>Avalon Memory Mapped Slave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m0</td>
<td>Avalon Memory Mapped Master</td>
</tr>
</tbody>
</table>

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 to ensure proper synchronization.
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.
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 different 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.
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.

**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**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Added: <em>Reset Polarity and Synchronization in Qsys.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime.</em></td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Multiplexer Examples, rearranged description text for the figures.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>AMBA APB support.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>AMBA AXI4 support.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>AMBA AXI3 support.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>New document release.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
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 on page 8-3
add_interface_port on page 8-5
get_interfaces on page 8-7
get_interface_assignment on page 8-8
get_interface_assignments on page 8-9
get_interface_ports on page 8-10
get_interface_properties on page 8-11
get_interface_property on page 8-12
get_port_properties on page 8-13
get_port_property on page 8-14
set_interface_assignment on page 8-15
set_interface_property on page 8-17
set_port_property on page 8-18
set_interface_upgrade_map on page 8-19
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.
<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>avalon</td>
<td>master, slave</td>
</tr>
<tr>
<td>axi</td>
<td>master, slave</td>
</tr>
<tr>
<td>tristate_conduit</td>
<td>master, slave</td>
</tr>
<tr>
<td>avalon_streaming</td>
<td>source, sink</td>
</tr>
<tr>
<td>interrupt</td>
<td>sender, receiver</td>
</tr>
<tr>
<td>conduit</td>
<td>end</td>
</tr>
<tr>
<td>clock</td>
<td>source, sink</td>
</tr>
<tr>
<td>reset</td>
<td>source, sink</td>
</tr>
<tr>
<td>nios_custom_instruction</td>
<td>slave</td>
</tr>
</tbody>
</table>

**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**

```tcl
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**

```tcl
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
add_interface_port

- 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**

```tcl
get_interfaces
```

**Returns**
A list of the top-level interfaces exported from the system.

**Arguments**
No arguments.

**Example**

```tcl
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.blockdiagram.direction: If present, the direction of this interface in the block diagram is set by the user. The value is either "output" or "input".
set_interface_assignment

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**

```tcl
set_interface_upgrade_map { <old_interface_name> <new_interface_name>
<old_interface_name_2> <new_interface_name_2> ... }
```

**Returns**
No return value.

**Arguments**

```tcl
{ <old_interface_name> <new_interface_name>}
```
List of mappings between between names of older and newer interfaces.

**Example**

```tcl
set_interface_upgrade_map { avalon_master_interface new_avalon_master_interface }
```
Parameters

add_parameter on page 8-21
get_parameters on page 8-22
get_parameter_properties on page 8-23
get_parameter_property on page 8-24
get_parameter_value on page 8-25
get_string on page 8-26
load_strings on page 8-28
set_parameter_property on page 8-29
set_parameter_value on page 8-30
decode_address_map on page 8-31
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**

```tcl
get_parameter_value <parameter>
```

**Returns**

The value of the parameter.

**Arguments**

```tcl
parameter
```

The name of the parameter whose value is being retrieved.

**Example**

```tcl
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 identifier.

**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`.

```tcl
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**

```tcl
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**

```tcl
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 on page 8-33
get_display_items on page 8-35
get_display_item_properties on page 8-36
get_display_item_property on page 8-37
set_display_item_property on page 8-38
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
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 on page 8-40
get_module_assignment on page 8-41
get_module_assignments on page 8-42
get_module_ports on page 8-43
get_module_properties on page 8-44
get_module_property on page 8-45
send_message on page 8-46
set_module_assignment on page 8-47
set_module_property on page 8-48
add_hdl_instance on page 8-49
package on page 8-50
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

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**

```tcl
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**

```tcl
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 on page 8-52
add_connection on page 8-52
get_connections on page 8-54
get_connection_parameters on page 8-55
get_connection_parameter_value on page 8-56
get_instances on page 8-57
get_instance_interfaces on page 8-58
get_instance_interface_ports on page 8-59
get_instance_interface_properties on page 8-60
get_instance_property on page 8-61
set_instance_property on page 8-62
get_instance_properties on page 8-63
get_instance_interface_property on page 8-64
get_instance_parameters on page 8-65
get_instance_parameter_property on page 8-66
get_instance_parameter_value on page 8-67
get_instance_port_property on page 8-68
set_connection_parameter_value on page 8-69
set_instance_parameter_value on page 8-70
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**

```tcl
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_connection 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 on page 8-72
add_fileset_file on page 8-73
set_fileset_property on page 8-74
get_fileset_file_attribute on page 8-75
set_fileset_file_attribute on page 8-76
get_fileset_properties on page 8-77
get_fileset_property on page 8-78
get_fileset_sim_properties on page 8-79
set_fileset_sim_properties on page 8-80
create_temp_file on page 8-81
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_filesset_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_filesset_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_filesset_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_filesset_file "./implementation/rx_pma.sv" SYSTEM_VERILOG PATH synth_rx_pma.sv
add_filesset_file gui.sv SYSTEM_VERILOG TEXT "Customize your IP core"

Related Information

- add_filesset on page 8-72
- get_filesset_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**

```tcl
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 on page 8-83
- get_device_family_displayname on page 8-84
- get_qip_strings on page 8-85
- set_qip_strings on page 8-86
- set_interconnect_requirement on page 8-87
check_device_family_equivalence

Description

Returns 1 if the device family is equivalent to one of the families in the device families list. 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**

```tcl
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 on page 8-89
Interface Properties on page 8-90
Instance Properties on page 8-91
Parameter Properties on page 8-92
Parameter Type Properties on page 8-95
Parameter Status Properties on page 8-96
Port Properties on page 8-97
Direction Properties on page 8-99
Display Item Properties on page 8-100
Display Item Kind Properties on page 8-101
Display Hint Properties on page 8-102
Module Properties on page 8-103
Fileset Properties on page 8-105
Fileset Kind Properties on page 8-106
Callback Properties on page 8-107
File Attribute Properties on page 8-108
File Kind Properties on page 8-109
File Source Properties on page 8-110
Simulator Properties on page 8-111
Port VHDL Type Properties on page 8-112
System Info Type Properties on page 8-113
Design Environment Type Properties on page 8-115
Units Properties on page 8-116
Operating System Properties on page 8-117
Quartus.ini Type Properties on page 8-118
### Script Language Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCL</td>
<td>Implements the script in Tcl.</td>
</tr>
</tbody>
</table>
## Interface Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMSIS_SVD_FILE</td>
<td>Specifies the connection point’s associated CMSIS file.</td>
</tr>
<tr>
<td>CMSIS_SVD_VARIABLES</td>
<td>Defines the variables inside a .svd file.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>Specifies whether or not interface is enabled.</td>
</tr>
</tbody>
</table>
| EXPORT_OF             | For composed _hw1.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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLINSTANCE_GET_GENERATED_NAME</td>
<td>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.</td>
</tr>
<tr>
<td>HDLINSTANCE_USE_GENERATED_NAME</td>
<td>If true, instances added with the add_hdl_instance command are instructed to use unique auto-generated fixed names based on the parameterization.</td>
</tr>
<tr>
<td>SUPPRESS_ALL_INFO_MESSAGES</td>
<td>If true, allows you to suppress all Info messages that originate from a child instance.</td>
</tr>
<tr>
<td>SUPPRESS_ALL_WARNINGS</td>
<td>If true, allows you to suppress all warnings that originate from a child instance.</td>
</tr>
</tbody>
</table>
## Parameter Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>AFFECTS_ELABORATION</td>
<td>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.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AFFECTS_GENERATION</td>
<td>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.</td>
</tr>
<tr>
<td>Boolean</td>
<td>AFFECTS_VALIDATION</td>
<td>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.</td>
</tr>
<tr>
<td>String[]</td>
<td>ALLOWED_RANGES</td>
<td>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 (&quot;dev1:Cyclone IV GX&quot;&quot;dev2:Stratix V GT&quot;).</td>
</tr>
<tr>
<td>String</td>
<td>DEFAULT_VALUE</td>
<td>The default value.</td>
</tr>
<tr>
<td>Boolean</td>
<td>DERIVED</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>DESCRIPTION</td>
<td>A short user-visible description of the parameter, suitable for a tooltip description in the parameter editor.</td>
</tr>
<tr>
<td>String[]</td>
<td>DISPLAY_HINT</td>
<td>Provides a hint about how to display a property. The following values are possible:</td>
</tr>
</tbody>
</table>

---

Altera Corporation
### Parameter Properties

#### 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.

<table>
<thead>
<tr>
<th>String</th>
<th>DISPLAY_NAME</th>
<th>This is the GUI label that appears to the left of this parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>DISPLAY_UNITS</td>
<td>This is the GUI label that appears to the right of the parameter.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>When false, the parameter is disabled, meaning that it is displayed, but greyed out, indicating that it is not editable on the parameter editor.</td>
</tr>
<tr>
<td>String</td>
<td>GROUP</td>
<td>Controls the layout of parameters in GUI.</td>
</tr>
<tr>
<td>Boolean</td>
<td>HDL_PARAMETER</td>
<td>When true, the parameter must be passed to the HDL IP component description. The default value is false.</td>
</tr>
<tr>
<td>String</td>
<td>LONG_DESCRIPTION</td>
<td>A user-visible description of the parameter. Similar to DESCRIPTION, but allows for a more detailed explanation.</td>
</tr>
<tr>
<td>String</td>
<td>NEW_INSTANCE_VALUE</td>
<td>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.</td>
</tr>
<tr>
<td>String[]</td>
<td>SYSTEM_INFO</td>
<td>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, &lt;info-type&gt;.</td>
</tr>
<tr>
<td>String</td>
<td>SYSTEM_INFO_ARG</td>
<td>Defines an argument to be passed to a particular SYSTEM_INFO function, such as the name of a reset interface.</td>
</tr>
<tr>
<td>(various)</td>
<td>SYSTEM_INFO_TYPE</td>
<td>Specifies one of the types of system information that can be queried. Refer to System Info Type Properties.</td>
</tr>
<tr>
<td>(various)</td>
<td>TYPE</td>
<td>Specifies the type of the parameter. Refer to Parameter Type Properties.</td>
</tr>
<tr>
<td>(various)</td>
<td>UNITS</td>
<td>Sets the units of the parameter. Refer to Units Properties.</td>
</tr>
<tr>
<td>Boolean</td>
<td>VISIBLE</td>
<td>Indicates whether or not to display the parameter in the parameterization GUI.</td>
</tr>
<tr>
<td>String</td>
<td>WIDTH</td>
<td>For a STD_LOGIC_VECTOR parameter, this indicates the width of the logic vector.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>A boolean parameter whose value is <code>true</code> or <code>false</code>.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>A signed 32-bit floating point parameter. Not supported for HDL parameters.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>A signed 32-bit integer parameter.</td>
</tr>
<tr>
<td>INTEGER_LIST</td>
<td>A parameter that contains a list of 32-bit integers. Not supported for HDL parameters.</td>
</tr>
<tr>
<td>LONG</td>
<td>A signed 64-bit integer parameter. Not supported for HDL parameters.</td>
</tr>
<tr>
<td>NATURAL</td>
<td>A 32-bit number that contains values 0 to 2147483647 (0x7fffffff).</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>A 32-bit number that contains values 1 to 2147483647 (0x7fffffff).</td>
</tr>
<tr>
<td>STD_LOGIC</td>
<td>A single bit parameter whose value can be 1 or 0;</td>
</tr>
<tr>
<td>STD_LOGIC VECTOR</td>
<td>An arbitrary-width number. The parameter property <code>WIDTH</code> determines the size of the logic vector.</td>
</tr>
<tr>
<td>STRING</td>
<td>A string parameter.</td>
</tr>
<tr>
<td>STRING_LIST</td>
<td>A parameter that contains a list of strings. Not supported for HDL parameters.</td>
</tr>
</tbody>
</table>
## Parameter Status Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>ACTIVE</td>
<td>Indicates the parameter is a regular parameter.</td>
</tr>
<tr>
<td>Boolean</td>
<td>DEPRECATED</td>
<td>Indicates the parameter exists only for backwards compatibility, and may not have any effect.</td>
</tr>
<tr>
<td>Boolean</td>
<td>EXPERIMENTAL</td>
<td>Indicates the parameter is experimental, and not exposed in the design flow.</td>
</tr>
</tbody>
</table>
## Port Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(various)</td>
<td>DIRECTION</td>
<td>The direction of the port from the IP component’s perspective. Refer to Direction Properties.</td>
</tr>
<tr>
<td>String</td>
<td>DRIVEN_BY</td>
<td>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.</td>
</tr>
<tr>
<td>String[]</td>
<td>FRAGMENT_LIST</td>
<td>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 &lt;interface&gt; foo &lt;role&gt; &lt;direction&gt; &lt;width&gt; add_interface_port &lt;interface&gt; bar &lt;role&gt; &lt;direction&gt; &lt;width&gt; set_port_property foo fragment_list &quot;my_rtl_signal(3:0)&quot; set_port_property bar fragment_list &quot;my_rtl_signal(6:4)&quot; Second you can take multiple RTL signals and combine them into a single Qsys signal: add_interface_port &lt;interface&gt; baz &lt;role&gt; &lt;direction&gt; &lt;width&gt; set_port_property baz fragment_list &quot;rtl_signal_1(3:0) rtl_signal_2(3:0)&quot; Note: The listed bits in a port fragment must match the declared width of the Qsys signal.</td>
</tr>
<tr>
<td>String</td>
<td>ROLE</td>
<td>Specifies an Avalon signal type such as waitrequest, readdata, or read. For a complete list of signal types, refer to the Avalon Interface Specifications.</td>
</tr>
<tr>
<td>Boolean</td>
<td>TERMINATION</td>
<td>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.</td>
</tr>
<tr>
<td>BigInteger</td>
<td>TERMINATION_VALUE</td>
<td>The constant value to drive an input port.</td>
</tr>
<tr>
<td>(various)</td>
<td>VHDL_TYPE</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>WIDTH</td>
<td>The width of the port in bits. Cannot be set directly. Any changes must be set through the WIDTH_EXPR property.</td>
</tr>
<tr>
<td>String</td>
<td>WIDTH_EXPR</td>
<td>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.</td>
</tr>
<tr>
<td>Integer</td>
<td>WIDTH_VALUE</td>
<td>The width of the port in bits. Cannot be set directly. Any changes must be set through the WIDTH_EXPR property.</td>
</tr>
</tbody>
</table>

### Related Information
- [Direction Properties](#) on page 8-99
Port Properties

- Port VHDL Type Properties on page 8-112
- Avalon Interface Specifications
### Direction Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bidir</td>
<td>Direction for a bidirectional signal.</td>
</tr>
<tr>
<td>Input</td>
<td>Direction for an input signal.</td>
</tr>
<tr>
<td>Output</td>
<td>Direction for an output signal.</td>
</tr>
</tbody>
</table>
## Display Item Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>DESCRIPTION</td>
<td>A description of the display item, which you can use as a tooltip.</td>
</tr>
<tr>
<td>String[]</td>
<td>DISPLAY_HINT</td>
<td>A hint that affects how the display item displays in the parameter editor.</td>
</tr>
<tr>
<td>String</td>
<td>DISPLAY_NAME</td>
<td>The label for the display item in the parameter editor.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>Indicates whether the display item is enabled or disabled.</td>
</tr>
<tr>
<td>String</td>
<td>PATH</td>
<td>The path to a file. Only applies to display items of type ICON.</td>
</tr>
<tr>
<td>String</td>
<td>TEXT</td>
<td>Text associated with a display item. Only applies to display items of type TEXT.</td>
</tr>
<tr>
<td>Boolean</td>
<td>VISIBLE</td>
<td>Indicates whether this display item is visible or not.</td>
</tr>
</tbody>
</table>
## Display Item Kind Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>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.</td>
</tr>
<tr>
<td>GROUP</td>
<td>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.</td>
</tr>
<tr>
<td>ICON</td>
<td>A .gif, .jpg, or .png file.</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>A parameter in the instance.</td>
</tr>
<tr>
<td>TEXT</td>
<td>A block of text.</td>
</tr>
</tbody>
</table>
Display Hint Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_WIDTH</td>
<td>Bit width of a number</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Integer value either 0 or 1.</td>
</tr>
<tr>
<td>COLLAPSED</td>
<td>Indicates whether a group is collapsed when initially displayed.</td>
</tr>
<tr>
<td>COLUMNS</td>
<td>Number of columns in text field, for example, &quot;columns:N&quot;.</td>
</tr>
<tr>
<td>EDITABLE</td>
<td>Indicates whether a list of strings allows free-form text entry (editable combo box).</td>
</tr>
<tr>
<td>FILE</td>
<td>Indicates that the string is an optional file path, for example, &quot;file:jpg,png,gif&quot;.</td>
</tr>
<tr>
<td>FIXED_SIZE</td>
<td>Indicates a fixed size for a table or list.</td>
</tr>
<tr>
<td>GROW</td>
<td>if set, the widget can grow when the IP component is resized.</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>Indicates that the long integer is hexadecimal.</td>
</tr>
<tr>
<td>RADIO</td>
<td>Indicates that the range displays as radio buttons.</td>
</tr>
<tr>
<td>ROWS</td>
<td>Number of rows in text field, or visible rows in a table, for example, &quot;rows:N&quot;.</td>
</tr>
<tr>
<td>SLIDER</td>
<td>Range displays as slider.</td>
</tr>
<tr>
<td>TAB</td>
<td>if present for a group, the group displays in a tab</td>
</tr>
<tr>
<td>TABLE</td>
<td>if present for a group, the group must contain all list-type parameters, which display collectively in a single table.</td>
</tr>
<tr>
<td>TEXT</td>
<td>String is a text field with a limited character set, for example, &quot;text:A-Za-z0-9_&quot;.</td>
</tr>
<tr>
<td>WIDTH</td>
<td>width of a table column</td>
</tr>
</tbody>
</table>
## Module Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE_HDL</td>
<td>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.</td>
</tr>
<tr>
<td>AUTHOR</td>
<td>The IP component author.</td>
</tr>
<tr>
<td>COMPOSITION_CALLBACK</td>
<td>The name of the composition callback. If you define a composition callback, you cannot define the generation or elaboration callbacks.</td>
</tr>
<tr>
<td>DATASHEET_URL</td>
<td>Deprecated. Use add_documentation_link to provide documentation links.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>The description of the IP component, such as &quot;This IP component puts the shizzle in the frobnitz.&quot;</td>
</tr>
<tr>
<td>DISPLAY_NAME</td>
<td>The name to display when referencing the IP component, such as &quot;My Qsys IP Component&quot;.</td>
</tr>
<tr>
<td>EDITABLE</td>
<td>Indicates whether you can edit the IP component in the Component Editor.</td>
</tr>
<tr>
<td>ELABORATION_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>GENERATION_CALLBACK</td>
<td>The name for a custom generation callback.</td>
</tr>
<tr>
<td>GROUP</td>
<td>The group in the IP Catalog that includes this IP component.</td>
</tr>
<tr>
<td>ICON_PATH</td>
<td>A path to an icon to display in the IP component's parameter editor.</td>
</tr>
<tr>
<td>INSTANTIATE_IN_SYSTEM_MODULE</td>
<td>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.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>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.</td>
</tr>
<tr>
<td>MODULE_DIRECTORY</td>
<td>The directory in which the hw.tcl file exists.</td>
</tr>
<tr>
<td>MODULE_TCL_FILE</td>
<td>The path to the hw.tcl file.</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the IP component, such as my_qsys_component.</td>
</tr>
<tr>
<td>OPAQUE_ADDRESS_MAP</td>
<td>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</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PREFERRED_SIMULATION_LANGUAGE</td>
<td>The preferred language to use for selecting the fileset for simulation model generation.</td>
</tr>
<tr>
<td>REPORT_HIERARCHY</td>
<td>null</td>
</tr>
<tr>
<td>STATIC_TOP_LEVEL_MODULE_NAME</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>STRUCTURAL_COMPOSITION_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>SUPPORTED_DEVICE_FAMILIES</td>
<td>A list of device family supported by this IP component.</td>
</tr>
<tr>
<td>TOP_LEVEL_HDL_FILE</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>TOP_LEVEL_HDL_MODULE</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>UPGRADEABLE_FROM</td>
<td>null</td>
</tr>
<tr>
<td>VALIDATION_CALLBACK</td>
<td>The name of the validation callback procedure.</td>
</tr>
<tr>
<td>VERSION</td>
<td>The IP component’s version, such as 10.0.</td>
</tr>
</tbody>
</table>
### Fileset Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLE_FILE_OVERWRITE_MODE</td>
<td>null</td>
</tr>
<tr>
<td>ENABLE_RELATIVE_INCLUDE_PATHS</td>
<td>If true, HDL files can include other files using relative paths in the fileset.</td>
</tr>
<tr>
<td>TOP_LEVEL</td>
<td>The name of the top-level HDL module that the fileset generates. If set, the HDL top level must match the <code>TOP_LEVEL</code> name, and the HDL must not be parameterized. Qsys runs the generate callback one time, regardless of the number of instances in the system.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EXAMPLE_DESIGN</td>
<td>Contains example design files.</td>
</tr>
<tr>
<td>QUARTUS_SYNTH</td>
<td>Contains files that Qsys uses for the Quartus Prime software synthesis.</td>
</tr>
<tr>
<td>SIM_VERILOG</td>
<td>Contains files that Qsys uses for Verilog HDL simulation.</td>
</tr>
<tr>
<td>SIM_VHDL</td>
<td>Contains files that Qsys uses for VHDL simulation.</td>
</tr>
</tbody>
</table>
Callback Properties

Description
This list describes each type of callback. Each command may only be available in some callback contexts.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>Called when an ACTION display item's action is performed.</td>
</tr>
<tr>
<td>COMPOSITION</td>
<td>Called during instance elaboration when the IP component contains a subsystem.</td>
</tr>
<tr>
<td>EDITOR</td>
<td>Called when the IP component is controlling the parameterization editor.</td>
</tr>
<tr>
<td>ELABORATION</td>
<td>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.</td>
</tr>
<tr>
<td>GENERATE_VERILOG_SIMULATION</td>
<td>Called when the IP component uses a custom generator to generates the Verilog simulation model for an instance.</td>
</tr>
<tr>
<td>GENERATE_VHDL_SIMULATION</td>
<td>Called when the IP component uses a custom generator to generates the VHDL simulation model for an instance.</td>
</tr>
<tr>
<td>GENERATION</td>
<td>Called when the IP component uses a custom generator to generates the synthesis HDL for an instance.</td>
</tr>
<tr>
<td>PARAMETER_UPGRADE</td>
<td>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.</td>
</tr>
<tr>
<td>STRUCTURAL_COMPOSITION</td>
<td>Called during instance elaboration when an IP component is represented by a structural hierarchical model which may be different from the generated RTL.</td>
</tr>
<tr>
<td>VALIDATION</td>
<td>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.</td>
</tr>
</tbody>
</table>
## File Attribute Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALDEC_SPECIFIC</td>
<td>Applies to Aldec simulation tools and for simulation filesets only.</td>
</tr>
<tr>
<td>CADENCE_SPECIFIC</td>
<td>Applies to Cadence simulation tools and for simulation filesets only.</td>
</tr>
<tr>
<td>COMMON_SYSTEMVERILOG_PACKAGE</td>
<td>The name of the common SystemVerilog package. Applies to simulation filesets only.</td>
</tr>
<tr>
<td>MENTOR_SPECIFIC</td>
<td>Applies to Mentor simulation tools and for simulation filesets only.</td>
</tr>
<tr>
<td>SYNOPSYS_SPECIFIC</td>
<td>Applies to Synopsys simulation tools and for simulation filesets only.</td>
</tr>
<tr>
<td>TOP_LEVEL_FILE</td>
<td>Contains the top-level module for the fileset and applies to synthesis filesets only.</td>
</tr>
</tbody>
</table>
## File Kind Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAT</td>
<td>DAT Data</td>
</tr>
<tr>
<td>FLI_LIBRARY</td>
<td>FLI Library</td>
</tr>
<tr>
<td>HEX</td>
<td>HEX Data</td>
</tr>
<tr>
<td>MIF</td>
<td>MIF Data</td>
</tr>
<tr>
<td>OTHER</td>
<td>Other</td>
</tr>
<tr>
<td>PLI_LIBRARY</td>
<td>PLI Library</td>
</tr>
<tr>
<td>QXP</td>
<td>QXP File</td>
</tr>
<tr>
<td>SDC</td>
<td>Timing Constraints</td>
</tr>
<tr>
<td>SYSTEM_VERILOG</td>
<td>System Verilog HDL</td>
</tr>
<tr>
<td>SYSTEM_VERILOG_ENCRYPT</td>
<td>Encrypted System Verilog HDL</td>
</tr>
<tr>
<td>SYSTEM_VERILOG_INCLUDE</td>
<td>System Verilog Include</td>
</tr>
<tr>
<td>VERILOG</td>
<td>Verilog HDL</td>
</tr>
<tr>
<td>VERILOG_ENCRYPT</td>
<td>Encrypted Verilog HDL</td>
</tr>
<tr>
<td>VERILOG_INCLUDE</td>
<td>Verilog Include</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHDL</td>
</tr>
<tr>
<td>VHDL_ENCRYPT</td>
<td>Encrypted VHDL</td>
</tr>
<tr>
<td>VPI_LIBRARY</td>
<td>VPI Library</td>
</tr>
</tbody>
</table>
File Source Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH</td>
<td>Specifies the original source file and copies to output_file.</td>
</tr>
<tr>
<td>TEXT</td>
<td>Specifies an arbitrary text string for the contents of output_file.</td>
</tr>
</tbody>
</table>
## Simulator Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENV_ALDEC_LD_LIBRARY_PATH</td>
<td>LD_LIBRARY_PATH when running riviera-pro</td>
</tr>
<tr>
<td>ENV_CADENCE_LD_LIBRARY_PATH</td>
<td>LD_LIBRARY_PATH when running ncsim</td>
</tr>
<tr>
<td>ENV_MENTOR_LD_LIBRARY_PATH</td>
<td>LD_LIBRARY_PATH when running modelsim</td>
</tr>
<tr>
<td>ENV_SYNOPSYS_LD_LIBRARY_PATH</td>
<td>LD_LIBRARY_PATH when running vcs</td>
</tr>
<tr>
<td>OPT_ALDEC_PLI</td>
<td>-pli option for riviera-pro</td>
</tr>
<tr>
<td>OPT_CADENCE_64BIT</td>
<td>-64bit option for ncsim</td>
</tr>
<tr>
<td>OPT_CADENCE_PLI</td>
<td>-loadpli1 option for ncsim</td>
</tr>
<tr>
<td>OPT_CADENCE_SVLIB</td>
<td>-sv_lib option for ncsim</td>
</tr>
<tr>
<td>OPT_CADENCE_SVROOT</td>
<td>-sv_root option for ncsim</td>
</tr>
<tr>
<td>OPT_MENTOR_64</td>
<td>-64 option for modelsim</td>
</tr>
<tr>
<td>OPT_MENTOR_CPPPATH</td>
<td>-cpppath option for modelsim</td>
</tr>
<tr>
<td>OPT_MENTOR_LDFLAGS</td>
<td>-ldflags option for modelsim</td>
</tr>
<tr>
<td>OPT_MENTOR_PLI</td>
<td>-pli option for modelsim</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_ACC</td>
<td>+acc option for vcs</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_CPP</td>
<td>-cpp option for vcs</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_FULL64</td>
<td>-full64 option for vcs</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_LDFLAGS</td>
<td>-LDFLAGS option for vcs</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_LLIB</td>
<td>-l option for vcs</td>
</tr>
<tr>
<td>OPT_SYNOPSYS_VPI</td>
<td>+vpi option for vcs</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AUTO</td>
<td>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.</td>
</tr>
<tr>
<td>STD_LOGIC</td>
<td>Indicates that the signal is not rendered in VHDL as a STD_LOGIC signal.</td>
</tr>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>Indicates that the signal is rendered in VHDL as a STD_LOGIC_VECTOR signal.</td>
</tr>
</tbody>
</table>
## System Info Type Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>ADDRESS_MAP</td>
<td>An XML-formatted string describing the address map for the interface specified in the system info argument.</td>
</tr>
<tr>
<td>Integer</td>
<td>ADDRESS_WIDTH</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>AVALON_SPEC</td>
<td>The version of the interconnect. SOPC Builder interconnect uses Avalon Specification 1.0. Qsys interconnect uses Avalon Specification 2.0.</td>
</tr>
<tr>
<td>Integer</td>
<td>CLOCK_DOMAIN</td>
<td>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.</td>
</tr>
<tr>
<td>Long, Integer</td>
<td>CLOCK_RATE</td>
<td>The rate of the clock connected to the clock input specified in the system info argument. If 0, the clock rate is currently unknown.</td>
</tr>
<tr>
<td>String</td>
<td>CLOCK_RESET_INFO</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>CUSTOM_INSTRUCTION_SLAVES</td>
<td>Provides custom instruction slave information, including the name, base address, address span, and clock cycle type.</td>
</tr>
<tr>
<td>(various)</td>
<td>DESIGN_ENVIRONMENT</td>
<td>A string that identifies the current design environment. Refer to Design Environment Type Properties.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE</td>
<td>The device part number of the currently selected device.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FAMILY</td>
<td>The family name of the currently selected device.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FEATURES</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_SPEEDGRADE</td>
<td>The speed grade of the currently selected device.</td>
</tr>
<tr>
<td>Integer</td>
<td>GENERATION_ID</td>
<td>A integer that stores a hash of the generation time to be used as a unique ID for a generation run.</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BigInteger, Long</td>
<td>INTERRUPTS_USED</td>
<td>A mask indicating which bits of an interrupt receiver are connected to interrupt senders. The interrupt receiver is specified in the system info argument.</td>
</tr>
<tr>
<td>Integer</td>
<td>MAX_SLAVE_DATA_WIDTH</td>
<td>The data width of the widest slave connected to the specified memory-mapped master.</td>
</tr>
<tr>
<td>String, Boolean, Integer</td>
<td>QUARTUS_INI</td>
<td>The value of the quartus.ini setting specified in the system info argument.</td>
</tr>
<tr>
<td>Integer</td>
<td>RESET_DOMAIN</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>TRISTATECONDUIT_INFO</td>
<td>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.</td>
</tr>
<tr>
<td>String</td>
<td>TRISTATECONDUIT_MASTERS</td>
<td>The names of the instance's interfaces that are tri-state conduit slaves.</td>
</tr>
<tr>
<td>String</td>
<td>UNIQUE_ID</td>
<td>A string guaranteed to be unique to this instance.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE</td>
<td>Design environment prefers native IP interfaces.</td>
</tr>
<tr>
<td>QSYS</td>
<td>Design environment prefers standard Qsys interfaces.</td>
</tr>
</tbody>
</table>
### Units Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>A memory-mapped address.</td>
</tr>
<tr>
<td>Bits</td>
<td>Memory size, in bits.</td>
</tr>
<tr>
<td>BitsPerSecond</td>
<td>Rate, in bits per second.</td>
</tr>
<tr>
<td>Bytes</td>
<td>Memory size, in bytes.</td>
</tr>
<tr>
<td>Cycles</td>
<td>A latency or count, in clock cycles.</td>
</tr>
<tr>
<td>GigabitsPerSecond</td>
<td>Rate, in gigabits per second.</td>
</tr>
<tr>
<td>Gigabytes</td>
<td>Memory size, in gigabytes.</td>
</tr>
<tr>
<td>Gigahertz</td>
<td>Frequency, in GHz.</td>
</tr>
<tr>
<td>Hertz</td>
<td>Frequency, in Hz.</td>
</tr>
<tr>
<td>KilobitsPerSecond</td>
<td>Rate, in kilobits per second.</td>
</tr>
<tr>
<td>Kilobytes</td>
<td>Memory size, in kilobytes.</td>
</tr>
<tr>
<td>Kilohertz</td>
<td>Frequency, in kHz.</td>
</tr>
<tr>
<td>MegabitsPerSecond</td>
<td>Rate, in megabits per second.</td>
</tr>
<tr>
<td>Megabytes</td>
<td>Memory size, in megabytes.</td>
</tr>
<tr>
<td>Megahertz</td>
<td>Frequency, in MHz.</td>
</tr>
<tr>
<td>Microseconds</td>
<td>Time, in micros.</td>
</tr>
<tr>
<td>Milliseconds</td>
<td>Time, in ms.</td>
</tr>
<tr>
<td>Nanoseconds</td>
<td>Time, in ns.</td>
</tr>
<tr>
<td>None</td>
<td>Unspecified units.</td>
</tr>
<tr>
<td>Percent</td>
<td>A percentage.</td>
</tr>
<tr>
<td>Picoseconds</td>
<td>Time, in ps.</td>
</tr>
<tr>
<td>Seconds</td>
<td>Time, in s.</td>
</tr>
</tbody>
</table>
### Operating System Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All operating systems</td>
</tr>
<tr>
<td>LINUX32</td>
<td>Linux 32-bit</td>
</tr>
<tr>
<td>LINUX64</td>
<td>Linux 64-bit</td>
</tr>
<tr>
<td>WINDOW32</td>
<td>Windows 32-bit</td>
</tr>
<tr>
<td>WINDOW64</td>
<td>Windows 64-bit</td>
</tr>
</tbody>
</table>
### Quartus.ini Type Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLED</td>
<td>Returns 1 if the setting is turned on, otherwise returns 0.</td>
</tr>
<tr>
<td>STRING</td>
<td>Returns the string value of the .ini setting.</td>
</tr>
</tbody>
</table>
## Document Revision History

The table below indicates edits made to the `Component Interface Tcl Reference` content since its creation.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Edit to <code>add_fileset_file</code> command.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• <code>set_interface_upgrade_map</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Moved <em>Port Roles (Interface Signal Types)</em> section to Qsys Interconnect.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>• <code>add_hdl_instance</code></td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>• Consolidated content from other Qsys chapters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added AMBA APB support.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Added the <code>demo_axi_memory</code> example with screen shots and example <code>_hw.tcl</code> code.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Added AMBA AXI3 support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <code>set_display_item_property</code>, <code>set_parameter_property</code>, <code>LONG_DESCRIPTION</code>, and static filesets.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>• Template update.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <code>set_qip_strings</code>, <code>get_qip_strings</code>, <code>get_device_family_displayname</code>, <code>check_device_family_equivalence</code>.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>• Revised section describing HDL and composed component implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Separated reset and clock interfaces in example.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <code>TRISTATECONDUIT_INFO</code>, <code>GENERATION_ID</code>, <code>UNIQUE_ID</code>, <code>SYSTEM_INFO</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <code>WIDTH</code> and <code>SYSTEM_INFO_ARG</code> parameter properties.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed the <code>doc_type</code> argument from the <code>add_documentation_link</code> command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed: <code>get_instance_parameter_properties</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added: <code>add_fileset</code>, <code>add_fileset_file</code>, <code>create_temp_file</code>.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Initial release.</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**

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

![Diagram of Clock Bridge](image)

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

- CPU (M)
- Avalon-MM Clock-Crossing Bridge (S)
- Avalon-MM Master Port
- Avalon-MM Slave Port
- DDR SDRAM
- Flash Memory
- JTAG Debug Module
- UART
- System ID
- Seven Segment PIO
- LCD Display
- External SRAM
- Avalon Tristate Bridge

Legend:
- M: Avalon-MM Master Port
- S: Avalon-MM Slave Port
# Avalon-MM Clock Crossing Bridge Parameters

## Table 9-1: Avalon-MM Clock Crossing Bridge Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024 bits</td>
<td>Determines the data width of the interfaces on the bridge, and affects the size of both FIFOs. For the highest bandwidth, set <strong>Data width</strong> to be as wide as the widest master that connects to the bridge.</td>
</tr>
<tr>
<td>Symbol width</td>
<td>1, 2, 4, 8, 16, 32, 64 (bits)</td>
<td>Number of bits per symbol. For example, byte-oriented interfaces have 8-bit symbols.</td>
</tr>
<tr>
<td>Address width</td>
<td>1-32 bits</td>
<td>The address bits needed to address the downstream slaves.</td>
</tr>
<tr>
<td>Use automatically-determined address width</td>
<td>-</td>
<td>The minimum bridge address width that is required to address the downstream slaves.</td>
</tr>
<tr>
<td>Maximum burst size</td>
<td>1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 bits</td>
<td>Determines the maximum length of bursts that the bridge supports.</td>
</tr>
<tr>
<td>Command FIFO depth</td>
<td>2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 bits</td>
<td>Command (master-to-slave) FIFO depth.</td>
</tr>
<tr>
<td>Respond FIFO depth</td>
<td>2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 bits</td>
<td>Slave-to-master FIFO depth.</td>
</tr>
<tr>
<td>Master clock domain synchronizer depth</td>
<td>2, 3, 4, 5 bits</td>
<td>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.</td>
</tr>
</tbody>
</table>

---

**Altera Corporation**

**Qsys System Design Components**

**Send Feedback**
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave clock domain synchronizer depth</td>
<td>2, 3, 4, 5 bits</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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.

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.
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

Table 9-2: Avalon-MM Unaligned Burst Expansion Bridge Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>Data width of the master connected to the bridge.</td>
</tr>
<tr>
<td>Address width (in WORDS)</td>
<td>The address width of the master connected to the bridge.</td>
</tr>
<tr>
<td>Burstcount width</td>
<td>The burstcount signal width of the master connected to the bridge.</td>
</tr>
</tbody>
</table>
### Parameter and Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum pending read transactions</strong></td>
<td>The <strong>Maximum pending read transactions</strong> 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.</td>
</tr>
<tr>
<td><strong>Width of slave to optimize for</strong></td>
<td>The data width of the connected slave. Supported values are: 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 bits.</td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>If you connect multiple slaves, all slaves must have the same data width.</td>
</tr>
<tr>
<td><strong>Pipeline command signals</strong></td>
<td>When turned on, the command path is pipelined, minimizing the bridge’s critical path at the expense of increased logic usage and latency.</td>
</tr>
</tbody>
</table>

### 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 $f_{\text{Max}}$ 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 interconnect 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.
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

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>AXI3</th>
<th>AXI4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>awid / arid</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awaddr /araddr</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awlen / arlen</code></td>
<td>yes (4-bit)</td>
<td>yes (8-bit)</td>
</tr>
<tr>
<td><code>awsize/ arsize</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awburst /arburst</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awlock /arlock</code></td>
<td>yes</td>
<td>yes (1-bit optional)</td>
</tr>
<tr>
<td><code>awcache / arcache</code></td>
<td>yes (2-bit)</td>
<td>yes (optional)</td>
</tr>
<tr>
<td><code>awprot / arprot</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awuser /aruser</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awvalid / arvalid</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awready /arready</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>awqos /arqos</code></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><code>awregion /arregion</code></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><code>wid</code></td>
<td>yes</td>
<td>no (optional)</td>
</tr>
<tr>
<td><code>wdata / rdata</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>wstrb</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>wlast /rvalid</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>wvalid /rlast</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>wready /rready</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>wuser / ruser</code></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><code>bid / rid</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>bresp / rresp</code></td>
<td>yes</td>
<td>yes (optional)</td>
</tr>
<tr>
<td><code>bvalid</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><code>bready</code></td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI Version</td>
<td>string</td>
<td>AXI3/AXI4</td>
<td>Specifies the AXI version and signals that Qsys generates for the slave and master interfaces of the bridge.</td>
</tr>
<tr>
<td>Data Width</td>
<td>integer</td>
<td>8:1024</td>
<td>Controls the width of the data for the master and slave interfaces.</td>
</tr>
<tr>
<td>Address Width</td>
<td>integer</td>
<td>1-64 bits</td>
<td>Controls the width of the address for the master and slave interfaces.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Range</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AWUSER Width</td>
<td>integer</td>
<td>1-64 bits</td>
<td>Controls the width of the write address channel sideband signals of the master and slave interfaces.</td>
</tr>
<tr>
<td>ARUSER Width</td>
<td>integer</td>
<td>1-64 bits</td>
<td>Controls the width of the read address channel sideband signals of the master and slave interfaces.</td>
</tr>
<tr>
<td>WUSER Width</td>
<td>integer</td>
<td>1-64 bits</td>
<td>Controls the width of the write data channel sideband signals of the master and slave interfaces.</td>
</tr>
<tr>
<td>RUSER Width</td>
<td>integer</td>
<td>1-16 bits</td>
<td>Controls the width of the read data channel sideband signals of the master and slave interfaces.</td>
</tr>
<tr>
<td>BUSER Width</td>
<td>integer</td>
<td>1-16 bits</td>
<td>Controls the width of the write response channel sideband signals of the master and slave interfaces.</td>
</tr>
</tbody>
</table>

**AXI Bridge Slave and Master Interface Parameters**

**Table 9-5: AXI Bridge Slave and Master Interface Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Width</td>
<td>Controls the width of the thread ID of the master and slave interfaces.</td>
</tr>
<tr>
<td>Write/Read/Combined Acceptance Capability</td>
<td>Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the slave interface accepts.</td>
</tr>
<tr>
<td>Write/Read/Combined Issuing Capability</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 backpressure.
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.
The AXI Timeout Bridge is notified that the slave is reset.

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

<table>
<thead>
<tr>
<th>Channel</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>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).</td>
<td>When the response is issued. First cycle of bvalid.</td>
</tr>
<tr>
<td>Read</td>
<td>When an address is issued. First cycle of arvalid.</td>
<td>When the last data is issued. First cycle of rvalid and rlast.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Attribute</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>write-only</td>
<td>Slave is reset</td>
<td>Write a 1 to notify the AXI Timeout Bridge that the slave is reset and ready. Clears the interrupt.</td>
</tr>
<tr>
<td>0x4</td>
<td>read-only</td>
<td>Timed out operation</td>
<td>The operation of the burst that caused the timeout. 1 for a write; 0 for a read.</td>
</tr>
<tr>
<td>0x8 through 0xf</td>
<td>read-only</td>
<td>Timed out address</td>
<td>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.</td>
</tr>
</tbody>
</table>

AXI Timeout Bridge Parameters

Table 9-8: AXI Timeout Bridge Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID width</td>
<td>The width of awid, bid, arid, or rid.</td>
</tr>
<tr>
<td>Address width</td>
<td>The width of awaddr or araddr.</td>
</tr>
<tr>
<td>Data width</td>
<td>The width of wdata or rdata.</td>
</tr>
<tr>
<td>User width</td>
<td>The width of awuser, wuser, buser, aruser, or ruser.</td>
</tr>
<tr>
<td>Maximum number of outstanding writes</td>
<td>The expected maximum number of outstanding writes.</td>
</tr>
<tr>
<td>Maximum number of outstanding reads</td>
<td>The expected maximum number of outstanding reads.</td>
</tr>
<tr>
<td>Maximum number of cycles</td>
<td>The number of cycles within which a burst must complete.</td>
</tr>
</tbody>
</table>

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.
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**

**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 0x110000. The slave appears to NIOS II as having an 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

Table 9-9: AXI Default Slave Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI master ID width</td>
<td>1-8 bits</td>
<td>Determines the master ID width for error logging.</td>
</tr>
<tr>
<td>AXI address width</td>
<td>8-64 bits</td>
<td>Determines the address width for error logging. This value also affects the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>overall address width of the system, and should not exceed the maximum</td>
</tr>
<tr>
<td>AXI data width</td>
<td>32, 64, or 128 bits</td>
<td>Determines the data width for error logging.</td>
</tr>
<tr>
<td>Enable CSR Support (for error logging)</td>
<td>On or Off</td>
<td>When turned on, instantiates an Avalon CSR interface for error logging.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CSR Error Log Depth</td>
<td>1-16 bits</td>
<td>Depth of the transaction log, for example, the number of transactions the CSR logs for cycles with errors.</td>
</tr>
<tr>
<td>Register Avalon CSR inputs</td>
<td>On or Off</td>
<td>When turned on, controls debug access to the CSR interface.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31:4</td>
<td>R0</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3</td>
<td>RW1C</td>
<td></td>
<td>0</td>
<td><strong>Read Access Violation Interrupt Overflow register</strong> 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.</td>
</tr>
<tr>
<td>2</td>
<td>RW1C</td>
<td></td>
<td>0</td>
<td><strong>Write Access Violation Interrupt Overflow register</strong> 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.</td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>31:13</td>
<td>R0</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>12:11</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst type of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>10:7</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst length of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>6:4</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst size of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>3:1</td>
<td>R0</td>
<td>0</td>
<td>Indicates the <strong>PROT</strong> of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>R0</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>0x104</td>
<td>31:0</td>
<td>R0</td>
<td>0</td>
<td>Master ID for the cycle that causes the access violation.</td>
</tr>
<tr>
<td>0x108</td>
<td>31:0</td>
<td>R0</td>
<td>0</td>
<td>Read cycle target address for the cycle that causes the access violation (lower 32-bit).</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x190</td>
<td>31:13</td>
<td>R0</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>12:11</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst type of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>10:7</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst length of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>6:4</td>
<td>R0</td>
<td>0</td>
<td>Indicates the burst size of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>3:1</td>
<td>R0</td>
<td>0</td>
<td>Indicates the PROT of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>R0</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>0x194</td>
<td>31:0</td>
<td>R0</td>
<td>0</td>
<td>Master ID for the cycle that causes the access violation.</td>
</tr>
<tr>
<td>0x198</td>
<td>31:0</td>
<td>R0</td>
<td>0</td>
<td>Write target address for the cycle that causes the access violation (lower 32-bit).</td>
</tr>
<tr>
<td>0x19C</td>
<td>31:0</td>
<td>R0</td>
<td>0</td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bit</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x1A0  | 31:0| R0        | 0       | First 32 bits of the write data for the write cycle that causes the access violation.  
**Note:** When this register is read, the current write access violation log is recovered from FIFO, when the data width is 32 bits. |
| 0x1A4  | 31:0| R0        | 0       | Bits \([63:32]\) of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 32 -bits. |
| 0x1A8  | 31:0| R0        | 0       | Bits \([95:64]\) of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits. |
| 0x1AC  | 31:0| R0        | 0       | The first bits \([127:96]\) of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits.  
**Note:** When this register is read, the current write access violation log is recovered from FIFO. |
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.
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.
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 a 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.

**Note:** All tri-state conduit components are connected to a pin sharer must be in the same clock domain.
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: \( \text{Symbol Value} = \text{Symbol Position in Packet} \oplus \text{Data Error Mask} \). 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.
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 is 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.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0</td>
<td>status</td>
</tr>
<tr>
<td>base + 1</td>
<td>control</td>
</tr>
<tr>
<td>base + 2</td>
<td>fill</td>
</tr>
</tbody>
</table>

Table 9-14: Test Pattern Generator Status Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:0]</td>
<td>ID</td>
<td>RO</td>
<td>A constant value of 0x64.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>NUMCHANNELS</td>
<td>RO</td>
<td>The configured number of channels.</td>
</tr>
<tr>
<td>[30:24]</td>
<td>NUMSYMBOLS</td>
<td>RO</td>
<td>The configured number of symbols per beat.</td>
</tr>
<tr>
<td>[31]</td>
<td>SUPPORTPACKETS</td>
<td>RO</td>
<td>A value of 1 indicates data packet support.</td>
</tr>
</tbody>
</table>

Table 9-15: Test Pattern Generator Control Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>ENABLE</td>
<td>RW</td>
<td>Setting this bit to 1 enables the test pattern generator core.</td>
</tr>
<tr>
<td>[7:1]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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 | |  

| 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 | |  

### 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

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:0]</td>
<td>SIZE</td>
<td>RW</td>
<td>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.</td>
</tr>
<tr>
<td>[29:16]</td>
<td>CHANNEL</td>
<td>RW</td>
<td>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.</td>
</tr>
<tr>
<td>[30]</td>
<td>SOP</td>
<td>RW</td>
<td>Set this bit to 1 when sending the first segment in a packet. This bit is ignored when data packets are not supported.</td>
</tr>
<tr>
<td>[31]</td>
<td>EOP</td>
<td>RW</td>
<td>Set this bit to 1 when sending the last segment in a packet. This bit is ignored when data packets are not supported.</td>
</tr>
</tbody>
</table>

Table 9-19: cmd_hi Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:0]</td>
<td>SIGNALED ERROR</td>
<td>RW</td>
<td>Specifies the value to drive the error signal. A non-zero value creates a signalled error.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>DATA ERROR</td>
<td>RW</td>
<td>The output data is XORed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.</td>
</tr>
<tr>
<td>[24]</td>
<td>SUPPRESS SOP</td>
<td>RW</td>
<td>Set this bit to 1 to suppress the assertion of the startofpacket signal when the first segment in a packet is sent.</td>
</tr>
<tr>
<td>[25]</td>
<td>SUPPRESS EOP</td>
<td>RW</td>
<td>Set this bit to 1 to suppress the assertion of the endofpacket signal when the last segment in a packet is sent.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0</td>
<td>status</td>
</tr>
<tr>
<td>base + 1</td>
<td>control</td>
</tr>
<tr>
<td>Offset</td>
<td>Register Name</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------</td>
</tr>
<tr>
<td>base + 2</td>
<td></td>
</tr>
<tr>
<td>base + 3</td>
<td>Reserved</td>
</tr>
<tr>
<td>base + 4</td>
<td></td>
</tr>
<tr>
<td>base + 5</td>
<td>exception_descriptor</td>
</tr>
<tr>
<td>base + 6</td>
<td>indirect_select</td>
</tr>
<tr>
<td>base + 7</td>
<td>indirect_count</td>
</tr>
</tbody>
</table>

Table 9-21: Test Pattern Checker Status Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:0]</td>
<td>ID</td>
<td>RO</td>
<td>Contains a constant value of 0x65.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>NUMCHANNELS</td>
<td>RO</td>
<td>The configured number of channels.</td>
</tr>
<tr>
<td>[30:24]</td>
<td>NUMSYMBOLS</td>
<td>RO</td>
<td>The configured number of symbols per beat.</td>
</tr>
<tr>
<td>[31]</td>
<td>SUPPORTPACKETS</td>
<td>RO</td>
<td>A value of 1 indicates packet support.</td>
</tr>
</tbody>
</table>

Table 9-22: Test Pattern Checker Control Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>ENABLE</td>
<td>RW</td>
<td>Setting this bit to 1 enables the test pattern checker.</td>
</tr>
<tr>
<td>[7:1]</td>
<td>Reserved</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>[16:8]</td>
<td>THROTTLE</td>
<td>RW</td>
<td>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.</td>
</tr>
<tr>
<td>[17]</td>
<td>SOFT RESET</td>
<td>RW</td>
<td>When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.</td>
</tr>
<tr>
<td>[31:18]</td>
<td>Reserved</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>
If there is no exception, reading the `exception_descriptor` register bit register returns 0.

### Table 9-23: `exception_descriptor` Register Bits

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>DATA ERROR</td>
<td>RO</td>
<td>A value of 1 indicates that an error is detected in the incoming data.</td>
</tr>
<tr>
<td>[7:3]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[15:8]</td>
<td>SIGNALLED ERROR</td>
<td>RO</td>
<td>The value of the error signal.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[31:24]</td>
<td>CHANNEL</td>
<td>RO</td>
<td>The channel on which the exception was detected.</td>
</tr>
</tbody>
</table>

### Table 9-24: `indirect_select` Register Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bits Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7:0]</td>
<td>INDIRECT CHANNEL</td>
<td>RW</td>
<td>Specifies the channel number that applies to the INDIRECT PACKET COUNT, INDIRECT SYMBOL COUNT, and INDIRECT ERROR COUNT registers.</td>
</tr>
<tr>
<td>[15:8]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[31:16]</td>
<td>INDIRECT ERROR</td>
<td>RO</td>
<td>The number of data errors that occurred on the channel specified by INDIRECT CHANNEL.</td>
</tr>
</tbody>
</table>

### Table 9-25: `indirect_count` Register Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bits Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:0]</td>
<td>INDIRECT PACKET COUNT</td>
<td>RO</td>
<td>The number of data packets received on the channel specified by INDIRECT CHANNEL.</td>
</tr>
<tr>
<td>[31:16]</td>
<td>INDIRECT SYMBOL COUNT</td>
<td>RO</td>
<td>The number of symbols received on the channel specified by INDIRECT CHANNEL.</td>
</tr>
</tbody>
</table>
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()` on page 9-46
- `data_source_init()` on page 9-47
- `data_source_get_id()` on page 9-47
- `data_source_get_supports_packets()` on page 9-48
- `data_source_get_num_channels()` on page 9-48
- `data_source_get_symbols_per_cycle()` on page 9-48
- `data_source_get_enable()` on page 9-49
- `data_source_set_enable()` on page 9-49
- `data_source_get_throttle()` on page 9-50
- `data_source_set_throttle()` on page 9-50
- `data_source_is_busy()` on page 9-51
- `data_source_fill_level()` on page 9-51
- `data_source_send_data()` on page 9-51

### `data_source_reset()`

**Table 9-26: data_source_reset()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>void data_source_reset(alt_u32 base);</code></td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
<tr>
<td>Description</td>
<td>Resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.</td>
</tr>
</tbody>
</table>
### data_source_init()

**Table 9-27: data_source_init()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prototype</strong></td>
<td>int data_source_init(alt_u32 base, alt_u32 command_base);</td>
</tr>
<tr>
<td><strong>Thread-safe</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Include</strong></td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td></td>
<td>command_base—Base address of the command slave.</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>1—Initialization is successful.</td>
</tr>
<tr>
<td></td>
<td>0—Initialization is unsuccessful.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Performs the following operations to initialize the test pattern generator core:</td>
</tr>
<tr>
<td></td>
<td>• Resets and disables the test pattern generator core.</td>
</tr>
<tr>
<td></td>
<td>• Sets the maximum throttle.</td>
</tr>
<tr>
<td></td>
<td>• Clears all inserted errors.</td>
</tr>
</tbody>
</table>

### data_source_get_id()

**Table 9-28: data_source_get_id()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prototype</strong></td>
<td>int data_source_get_id(alt_u32 base);</td>
</tr>
<tr>
<td><strong>Thread-safe</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Include</strong></td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>Test pattern generator core identifier.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Retrieves the test pattern generator core’s identifier.</td>
</tr>
</tbody>
</table>
### data_source_get_supports_packets()

Table 9-29: data_source_get_supports_packets()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_init(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Data packets are supported. 0—Data packets are not supported.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if the test pattern generator core supports data packets.</td>
</tr>
</tbody>
</table>

### data_source_get_num_channels()

Table 9-30: data_source_get_num_channels()

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_get_num_channels(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of channels supported.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of channels supported by the test pattern generator core.</td>
</tr>
</tbody>
</table>

### data_source_get_symbols_per_cycle()

Table 9-31: data_source_get_symbols_per_cycle()

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_get_symbols(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of symbols transferred in a beat.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of symbols transferred by the test pattern generator core in each beat.</td>
</tr>
</tbody>
</table>

**data_source_get_enable()**

Table 9-32: data_source_get_enable()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>int data_source_get_enable(alt_u32 base);</code></td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Value of the <code>ENABLE</code> bit.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the value of the <code>ENABLE</code> bit.</td>
</tr>
</tbody>
</table>

**data_source_set_enable()**

Table 9-33: data_source_set_enable()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>void data_source_set_enable(alt_u32 base, alt_u32 value);</code></td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
</tbody>
</table>
| 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
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()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_get_throttle(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Throttle value.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the current throttle value.</td>
</tr>
</tbody>
</table>

### data_source_set_throttle()

**Table 9-35: data_source_set_throttle()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_source_set_throttle(alt_u32 base, alt_u32 value) ;</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave. value—Throttle value.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>
### data_source_is_busy()

Table 9-36: data_source_is_busy()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_is_busy(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Test pattern generator core is busy. 0—Test pattern generator core is not busy.</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

### data_source_fill_level()

Table 9-37: data_source_fill_level()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_fill_level(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_source_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of commands in the command FIFO.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of commands currently in the command FIFO.</td>
</tr>
</tbody>
</table>

### data_source_send_data()

Table 9-38: data_source_send_data()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>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);</td>
</tr>
<tr>
<td>Information Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_source_util.h&gt;</code></td>
</tr>
</tbody>
</table>

### 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_SUPPRESS_SOP` and `DATA_SOURCE_SEND_SUPPRESS_EOP`.
- **error** — Value asserted on the `error` signal on the output interface.
- **data_error_mask** — Parameter and the data are XORed 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_get_packet_count() on page 9-57
data_sink_get_error_count() on page 9-58
data_sink_get_symbol_count() on page 9-58
data_sink_get_exception() on page 9-58
data_sink_exception_is_exception() on page 9-59
data_sink_exception_has_data_error() on page 9-59
data_sink_exception_has_missing_sop() on page 9-60
data_sink_exception_has_missing_eop() on page 9-60
data_sink_exception_signalled_error() on page 9-61

data_sink_reset()

Table 9-39: data_sink_reset()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_sink_reset(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>Resets the test pattern checker core including all internal counters.</td>
</tr>
</tbody>
</table>

data_sink_init()

Table 9-40: data_sink_init()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_init(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
</tbody>
</table>
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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_id(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Test pattern checker core identifier.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the test pattern checker core’s identifier.</td>
</tr>
</tbody>
</table>

data_sink_get_supports_packets()

Table 9-42: data_sink_get_supports_packets()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_init(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Data packets are supported. 0—Data packets are not supported.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if the test pattern checker core supports data packets.</td>
</tr>
</tbody>
</table>
**data_sink_get_num_channels()**

Table 9-43: data_sink_get_num_channels()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_num_channels(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of channels supported.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of channels supported by the test pattern checker core.</td>
</tr>
</tbody>
</table>

**data_sink_get_symbols_per_cycle()**

Table 9-44: data_sink_get_symbols_per_cycle()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_symbols(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of symbols received in a beat.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of symbols received by the test pattern checker core in each beat.</td>
</tr>
</tbody>
</table>

**data_sink_get_enable()**

Table 9-45: data_sink_get_enable()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_enable(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### data_sink_set_enable()

Table 9-46: data_sink_set_enable()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_sink_set_enable(alt_u32 base, alt_u32 value);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h &gt;</td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_throttle(alt_u32 base);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h &gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>Throttle value.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the throttle value.</td>
</tr>
</tbody>
</table>
### data_sink_set_throttle()

**Table 9-48: data_sink_set_throttle()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_sink_set_throttle(alt_u32 base, alt_u32 value);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_error_count(alt_u32 base, alt_u32 channel);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_sink_util.h&gt;</code></td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>No</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_sink_util.h&gt;</code></td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_get_exception(alt_u32 base);</td>
</tr>
</tbody>
</table>
### data_sink_exception_is_exception()

**Table 9-53: data_sink_exception_is_exception()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>int data_sink_exception_is_exception(int exception);</code></td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_sink_util.h&gt;</code></td>
</tr>
<tr>
<td>Parameters</td>
<td><code>exception</code>—Exception descriptor</td>
</tr>
</tbody>
</table>
| 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()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>int data_sink_exception_has_data_error(int exception);</code></td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td><code>&lt;data_sink_util.h&gt;</code></td>
</tr>
<tr>
<td>Parameters</td>
<td><code>exception</code>—Exception descriptor.</td>
</tr>
</tbody>
</table>
data_sink_exception_has_missing_sop()

Table 9-55: data_sink_exception_has_missing_sop()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_exception_has_missing_sop(int exception);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>\textless\textit{data_sink_util.h}\textgreater</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_exception_has_missing_eop(int exception);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>\textless\textit{data_sink_util.h}\textgreater</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
</tbody>
</table>
| 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()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_exception_signalled_error(int exception);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
<tr>
<td>Returns</td>
<td>Signal error value.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the value of the signalled error from the exception.</td>
</tr>
</tbody>
</table>

**data_sink_exception_channel()**

Table 9-58: data_sink_exception_channel()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_sink_exception_channel(int exception);</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>Yes</td>
</tr>
<tr>
<td>Include</td>
<td>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
<tr>
<td>Returns</td>
<td>Channel number on which an exception occurred.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the channel number on which an exception occurred.</td>
</tr>
</tbody>
</table>
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 not 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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Ready latency = 0.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Configurable.</td>
</tr>
<tr>
<td>Channel</td>
<td>Supported (optional).</td>
</tr>
<tr>
<td>Error</td>
<td>Supported (optional).</td>
</tr>
<tr>
<td>Packet</td>
<td>Supported (optional).</td>
</tr>
</tbody>
</table>

### Splitter Core Parameters

#### Table 9-60: Avalon-ST Splitter Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Of Outputs</td>
<td>1 to 16</td>
<td>2</td>
<td>The number of output interfaces. Qsys supports 1 for some systems where no duplicated output is required.</td>
</tr>
<tr>
<td>Qualify Valid Out</td>
<td>0 or 1</td>
<td>1</td>
<td>Determines whether the out_valid signal is gated or non-gated when backpressure is applied.</td>
</tr>
<tr>
<td>Data Width</td>
<td>1–512</td>
<td>8</td>
<td>The width of the data on the Avalon-ST data interfaces.</td>
</tr>
<tr>
<td>Bits Per Symbol</td>
<td>1–512</td>
<td>8</td>
<td>The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.</td>
</tr>
<tr>
<td>Use Packets</td>
<td>0 or 1</td>
<td>0</td>
<td>Indicates whether or not data packet transfers are supported. Packet support includes the startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>Use Channel</td>
<td>0 or 1</td>
<td>0</td>
<td>The option to enable or disable the channel signal.</td>
</tr>
<tr>
<td>Channel Width</td>
<td>0–8</td>
<td>1</td>
<td>The width of the channel signal on the data interfaces. This parameter is disabled when Use Channel is set to 0.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Legal Values</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Max Channels</td>
<td>0-255</td>
<td>1</td>
<td>The maximum number of channels that a data interface can support. This parameter is disabled when Use Channel is set to 0.</td>
</tr>
<tr>
<td>Use Error</td>
<td>0 or 1</td>
<td>0</td>
<td>The option to enable or disable the error signal.</td>
</tr>
<tr>
<td>Error Width</td>
<td>0–31</td>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Not supported.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Configurable.</td>
</tr>
<tr>
<td>Channel</td>
<td>Supported (optional).</td>
</tr>
<tr>
<td>Error</td>
<td>Supported (optional).</td>
</tr>
<tr>
<td>Packet</td>
<td>Supported (optional).</td>
</tr>
</tbody>
</table>

**Delay Core Parameters**

**Table 9-62: Avalon-ST Delay Core Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Of Delay Clocks</td>
<td>0 to 16</td>
<td>1</td>
<td>Specifies the delay the core introduces, in clock cycles. Qsys supports 0 for some systems where no delay is required.</td>
</tr>
<tr>
<td>Data Width</td>
<td>1–512</td>
<td>8</td>
<td>The width of the data on the Avalon-ST data interfaces.</td>
</tr>
<tr>
<td>Bits Per Symbol</td>
<td>1–512</td>
<td>8</td>
<td>The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.</td>
</tr>
<tr>
<td>Use Packets</td>
<td>0 or 1</td>
<td>0</td>
<td>Indicates whether or not data packet transfers are supported. Packet support includes the startofpacket, endofpacket, and empty signals.</td>
</tr>
<tr>
<td>Use Channel</td>
<td>0 or 1</td>
<td>0</td>
<td>The option to enable or disable the channel signal.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Legal Values</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Channel Width</td>
<td>0-8</td>
<td>1</td>
<td>The width of the channel signal on the data interfaces. This parameter is disabled when Use Channel is set to 0.</td>
</tr>
<tr>
<td>Max Channels</td>
<td>0-255</td>
<td>1</td>
<td>The maximum number of channels that a data interface can support. This parameter is disabled when Use Channel is set to 0.</td>
</tr>
<tr>
<td>Use Error</td>
<td>0 or 1</td>
<td>0</td>
<td>The option to enable or disable the error signal.</td>
</tr>
<tr>
<td>Error Width</td>
<td>0–31</td>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Not supported</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data width = 1; Bits per symbol = 1</td>
</tr>
<tr>
<td>Channel</td>
<td>Maximum channel = 32; Channel width = 5</td>
</tr>
<tr>
<td>Error</td>
<td>Not supported</td>
</tr>
<tr>
<td>Packet</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

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 \( \text{waitrequest} \) when it cannot accept new requests.

Table 9-64: Avalon-ST Round Robin Scheduler Ports

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>In</td>
<td>Clock reference.</td>
</tr>
<tr>
<td>reset_n</td>
<td>In</td>
<td>Asynchronous active low reset.</td>
</tr>
</tbody>
</table>
### Avalon-MM Request Interface

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_address</td>
<td>Out</td>
<td>The write address that indicates which channel has the request.</td>
</tr>
<tr>
<td>request_write</td>
<td>Out</td>
<td>Write enable signal.</td>
</tr>
<tr>
<td>request_writedata</td>
<td>Out</td>
<td>The amount of data requested from the particular channel. This value is always fixed at 1.</td>
</tr>
<tr>
<td>request_waitrequest</td>
<td>In</td>
<td>Wait request signal that pauses the scheduler when the slave cannot accept a new request.</td>
</tr>
</tbody>
</table>

### Avalon-ST Almost-Full Status Interface

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>almost_full_valid</td>
<td>In</td>
<td>Indicates that almost_full_channel and almost_full_data are valid.</td>
</tr>
<tr>
<td>almost_full_channel</td>
<td>In</td>
<td>Indicates the channel for the current status indication.</td>
</tr>
<tr>
<td>almost_full_data</td>
<td>In</td>
<td>A 1-bit signal that is asserted high to indicate that the channel indicated by almost_full_channel is almost full.</td>
</tr>
</tbody>
</table>

### Round Robin Scheduler Parameters

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of channels</td>
<td>2–32</td>
<td>Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.</td>
</tr>
<tr>
<td>Use almost-full status</td>
<td>0–1</td>
<td>Specifies whether the scheduler uses the almost-full interface. If not, the core requests data from the next channel at the next clock cycle.</td>
</tr>
</tbody>
</table>
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.

**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**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Ready latency = 0.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Data width = 8 bits; Bits per symbol = 8.</td>
</tr>
<tr>
<td>Channel</td>
<td>Not supported.</td>
</tr>
<tr>
<td>Feature</td>
<td>Property</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>Error</td>
<td>Not used.</td>
</tr>
<tr>
<td>Packet</td>
<td>Supported.</td>
</tr>
</tbody>
</table>

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 in the table below.

**Table 9-67: Data Packet Formats**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Transaction Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Transaction code</td>
<td>Type of transaction.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>[3:2]</td>
<td>Size</td>
<td>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.</td>
</tr>
<tr>
<td>[n:8]</td>
<td>Data</td>
<td>Transaction data; data to be written for write transactions.</td>
</tr>
<tr>
<td></td>
<td><strong>Response Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Transaction code</td>
<td>The transaction code with the most significant bit inversed.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
Packets to Transactions Converter Supported Transactions

Table 9-68: Packets to Transactions Converter Supported Transactions

<table>
<thead>
<tr>
<th>Transaction Code</th>
<th>Avalon-MM Transaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Write, non-incrementing address.</td>
<td>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.</td>
</tr>
<tr>
<td>0x04</td>
<td>Write, incrementing address.</td>
<td>Writes transaction data starting at the current address.</td>
</tr>
<tr>
<td>0x10</td>
<td>Read, non-incrementing address.</td>
<td>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.</td>
</tr>
<tr>
<td>0x14</td>
<td>Read, incrementing address.</td>
<td>Reads the number of bytes specified in the size parameter starting from the current address.</td>
</tr>
<tr>
<td>0x7f</td>
<td>No transaction.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 processes packets without an end of packet (EOP).

- **Unsupported transaction codes** — The core processes unsupported transactions as a no transaction.

Avalon-ST Streaming Pipeline Stage

The Avalon-ST pipeline stage receives data from an Avalon-ST source interface, and outputs the data to an Avalon-ST sink interface. In the absence of back pressure, the Avalon-ST pipeline stage source interface outputs data one cycle after receiving the data on its sink interface.

If the pipeline stage receives back pressure on its source interface, it continues to assert its source interface’s current data output. While the pipeline stage is receiving back pressure on its source interface and it receives new data on its sink interface, the pipeline stage internally buffers the new data. It then asserts back pressure on its sink interface.

Once the backpressure is deasserted, the pipeline stage’s source interface is de-asserted and the pipeline stage asserts internally buffered data (if present). Additionally, the pipeline stage de-asserts back pressure on its sink interface.

**Figure 9-29: Pipeline Stage Simple Register**

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

```
| data_in | Sink | Register 0 | Source | data_out |
```
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.
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:

\[(\log_2 (n-1)) + 1 + w\]

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 \( \log_2(\text{num_output_interfaces}) \) 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 go to channel 0, and the odd channels go to channel 1. If the high-order bits of the channel signal select the output interface, channels 0 to 7 go to channel 0 and channels 8 to 15 go 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

- **Avalon-MM Slave**
- **Avalon-ST Data Sink**
- **Avalon-ST Single-Clock FIFO**
- **Avalon-ST Data Source**
- **Avalon-ST Status Source**
- **almost_full**
- **almost_empty**
- **in**
- **out**
- **csr**
Interaces 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**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backpressure</td>
<td>Ready latency = 0.</td>
</tr>
<tr>
<td>Data Width</td>
<td>Configurable.</td>
</tr>
</tbody>
</table>
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 $f_{\text{MAX}}$. 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
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per symbol</td>
<td>1–32</td>
<td>These parameters determine the width of the FIFO.</td>
</tr>
<tr>
<td>Symbols per beat</td>
<td>1–32</td>
<td>FIFO width = Bits per symbol * Symbols per beat, where:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits per symbol is the number of bits in a symbol, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Symbols per beat is the number of symbols transferred in a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>beat.</td>
</tr>
<tr>
<td>Error width</td>
<td>0–32</td>
<td>The width of the error signal.</td>
</tr>
<tr>
<td>FIFO depth</td>
<td>$2^n$</td>
<td>The FIFO depth. An output pipeline stage is added to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIFO to increase performance, which increases the FIFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>depth by one. $n = 1, 2, 3, 4...$</td>
</tr>
<tr>
<td>Use packets</td>
<td>—</td>
<td>Turn on this parameter to enable data packet support on the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Avalon-ST data interfaces.</td>
</tr>
<tr>
<td>Channel width</td>
<td>1–32</td>
<td>The width of the channel signal.</td>
</tr>
<tr>
<td>Avalon-ST Single Clock FIFO Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use fill level</td>
<td>—</td>
<td>Turn on this parameter to include the Avalon-MM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>control and status register interface (CSR). The CSR is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enabled when Use fill level is set to 1.</td>
</tr>
</tbody>
</table>

the input clock domain represents the amount of space available in the FIFO (available space = FIFO depth – input fill level).
### Single-Clock and Dual-Clock FIFO Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use Store and Forward</strong></td>
<td></td>
<td>To turn on <strong>Cut-through mode</strong>, <strong>Use store and forward</strong> must be set to 0. Turning on <strong>Use store and forward</strong> prompts the user to turn on <strong>Use fill level</strong>, and then the CSR appears.</td>
</tr>
<tr>
<td><strong>Avalon-ST Dual Clock FIFO Only</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Use sink fill level</strong></td>
<td></td>
<td>Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.</td>
</tr>
<tr>
<td><strong>Use source fill level</strong></td>
<td></td>
<td>Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.</td>
</tr>
<tr>
<td><strong>Write pointer synchronizer length</strong></td>
<td>2–8</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Read pointer synchronizer length</strong></td>
<td>2–8</td>
<td>The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.</td>
</tr>
<tr>
<td><strong>Use Max Channel</strong></td>
<td></td>
<td>Turn on this parameter to specify the maximum channel number.</td>
</tr>
<tr>
<td><strong>Max Channel</strong></td>
<td>1–255</td>
<td>Maximum channel number.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>32-Bit Word Offset</th>
<th>Name</th>
<th>Access</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fill_level</td>
<td>R</td>
<td>0</td>
<td>24-bit FIFO fill level. Bits 24 to 31 are not used.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>2</td>
<td>almost_full_threshold</td>
<td>RW</td>
<td>FIFO depth–1</td>
<td>Set this register to a value that indicates the FIFO buffer is getting full.</td>
</tr>
<tr>
<td>3</td>
<td>almost_empty_threshold</td>
<td>RW</td>
<td>0</td>
<td>Set this register to a value that indicates the FIFO buffer is getting empty.</td>
</tr>
<tr>
<td>4</td>
<td>cut_through_threshold</td>
<td>RW</td>
<td>0</td>
<td>0—Enables store and forward mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>5</td>
<td>drop_on_error</td>
<td>RW</td>
<td>0</td>
<td>0—Disables drop-on error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1—Enables drop-on error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This register applies only when the Use packet and Use store and forward parameters are turned on.</td>
</tr>
</tbody>
</table>

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.

### 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.

<table>
<thead>
<tr>
<th>32-Bit Word Offset</th>
<th>Name</th>
<th>Access</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fill_level</td>
<td>R</td>
<td>0</td>
<td>24-bit FIFO fill level. Bits 24 to 31 are not used.</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Avalon-MM Unaligned Burst Expansion Bridge and Avalon-MM Pipeline Bridge, Maximum pending read transactions parameter. Extended description.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• AXI Timeout Bridge.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added notes to <em>Avalon-MM Clock Crossing Bridge</em> pertaining to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SDC constraints for its internal asynchronous FIFOs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FIFO-based clock crossing.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• AXI Bridge support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Address Span Extender updates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Avalon-MM Unaligned Burst Expansion Bridge support.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>• Address Span Extender</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>• Added Streaming Pipeline Stage support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added AMBA APB support.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Moved relevant content from the <em>Embedded Peripherals IP User Guide</em>.</td>
</tr>
</tbody>
</table>

Altera Corporation
Related Information

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

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 combinational 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.
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.

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 Asynchronous 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 falling 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 \texttt{-rtl} option with the \texttt{quartus_drc} executable, as shown in the following example:

\texttt{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 \texttt{<project_name>.drc.rpt} in the \texttt{<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:

\texttt{<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**

\texttt{<DA_RULE ID="EX01" SEVERITY="CRITICAL" NAME="Checking Design for SR Latch" DEFAULT_RUN="YES">}
\texttt{<RULE_DEFINITION>}
\texttt{<FORBID>}
\texttt{<OR>}
\texttt{<NODE NAME="NODE_1" TYPE="SRLATCH" />}
\texttt{<HAS_NODE NODE_LIST="NODE_1" />}
\texttt{<NODE NAME="NODE_1" TOTAL_FANIN="EQ2" />}
\texttt{<NODE NAME="NODE_2" TOTAL_FANIN="EQ2" />}
\texttt{<AND>}
\texttt{<NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND"}
\texttt{TO_NAME="NODE_2" TO_TYPE="NAND" />}
\texttt{<NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND"}
\texttt{TO_NAME="NODE_1" TO_TYPE="NAND" />}
\texttt{</AND>}
\texttt{</OR>}
\texttt{</FORBID>}
\texttt{</RULE_DEFINITION>}
\texttt{</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>
```
This example shows how to use the \texttt{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**

```xml
<DA_RULE ID="EX02" SEVERITY="HIGH" NAME="Data Transfer Not Synch Correctly" DEFAULT_RUN="YES">
  <RULE_DEFINITION>
    <DECLARE>
      <NODE NAME="NODE_1" TYPE="REG" />
      <NODE NAME="NODE_2" TYPE="REG" />
      <NODE NAME="NODE_3" TYPE="REG" />
    </DECLARE>
    <FORBID>
      <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D\_PORT" CLOCK_RELATIONSHIP="ASYN" />
      <NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D\_PORT" CLOCK_RELATIONSHIP="!ASYN" />
      <OR>
        <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D\_PORT" REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
        <CLOCK_RELATIONSHIP NAME="SEQ\_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
      </OR>
    </FORBID>
  </RULE_DEFINITION>
  <REPORTING_ROOT>
    <MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
      <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
      <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL\_NODE" VALUE="NODE\_1" />
    </MESSAGE>
    <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>
  </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.

\[
\begin{align*}
&\text{<NODE_RELATIONSHIP FROM NAME="NODE_1" TO NAME="NODE_2" TO PORT="D_PORT" CLOCK RELATIONSHIP="ASYN"} /> \\
&\text{<NODE_RELATIONSHIP FROM NAME="NODE_2" TO NAME="NODE_3" TO PORT="D_PORT" CLOCK RELATIONSHIP="!ASYN"} /> \\
&\text{The next line of code states that NODE_2 and NODE_3 have a clock relationship of either sequential edge or asynchronous.} \\
&\text{<CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE LIST="NODE_2, NODE_3"} />
\end{align*}
\]

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**

\[
\begin{align*}
&\text{<NODE_RELATIONSHIP FROM NAME="NODE_1" TO NAME="NODE_2" TO PORT="D_PORT" CLOCK RELATIONSHIP="ASYN"} />\\
&\text{<NODE_RELATIONSHIP FROM NAME="NODE_2" TO NAME="NODE_3" TO PORT="D_PORT" CLOCK RELATIONSHIP="!ASYN"} />\\
&\text{<NODE_RELATIONSHIP FROM NAME="NODE_1" TO NAME="NODE_2" TO PORT="D_PORT" REQUIRED THROUGH="YES" THROUGH_TYPE="COMB" CLOCK RELATIONSHIP="ASYN"} />
\end{align*}
\]
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.
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.
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

```verilog
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

```verilog
# Input clock - 100 MHz
create_clock [get_ports {clock}] \
- name {clock} \n- period 10.0 \n- waveform {0.0 5.0}
# Input constraints on low-active reset and data
set_input_delay 7.0 \n -max \n -clock [get_clocks {clock}] \n [get_ports {reset_n data_a data_b}]
set_input_delay 1.0 \n -min \n -clock [get_clocks {clock}] \n [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

```verilog
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;
```

Use Synchronous Resets

10-22

QPPSV1
2015.11.02

Altera Corporation
Recommended Design Practices

Send Feedback
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**

```plaintext
# 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.
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<sub>su</sub>) or removal (µt<sub>H</sub>) 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.
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**

```verilog
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**

```verilog
# Input clock - 100 MHz
create_clock [get_ports {clock}] \
```

**Recommended Design Practices**

Altera Corporation
Use Synchronized Asynchronous Reset

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 instantly, 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.
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**

```verilog
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)
```
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 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 synchron-
ized 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**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Removed references to obsolete MegaWizard Plug-In Manager.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device information.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>Removed PrimeTime support.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Added information to Reset Resources.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>- Title changed from Design Recommendations for Altera Devices and the Quartus Prime Design Assistant.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated to new template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Removed duplicated content and added references to Quartus Prime Help for “Custom Rules” on page 9–15.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Removed information from “Combinational Logic Structures” on page 5–4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Changed heading from “Design Techniques to Save Power” to “Power Optimization” on page 5–12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new “Metastability” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new “Incremental Compilation” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added information to “Reset Resources” on page 5–23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Removed “Referenced Documents” section</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>- Removed documentation of obsolete rules.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>- No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>- Changed to 8-1/2 x 11 page size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section “Custom Rules Coding Examples” on page 5–18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added paragraph to “Recommended Clock-Gating Methods” on page 5–11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section: “Design Techniques to Save Power” on page 5–12</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>- Updated Figure 5–9 on page 5–13; added custom rules file to the flow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added notes to Figure 5–9 on page 5–13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section: “Custom Rules Report” on page 5–34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section: “Custom Rules” on page 5–34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section: “Targeting Embedded RAM Architectural Features” on page 5–38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Minor editorial updates throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
</tbody>
</table>

**Related Information**

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
- [link/mwh1409960181641/mwh1409959843979](link/mwh1409960181641/mwh1409959843979)
- [Recommended Design Practices](Recommended Design Practices) on page 10-1
- [Advanced Synthesis Cookbook](Advanced Synthesis Cookbook)
- [Design Examples](Design Examples)
- [Reference Designs](Reference Designs)

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**

---

**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

```verilog
module unsigned_mult (out, a, b);
  output [15:0] out;
  input [7:0] a;
  input [7:0] b;
```

---

**Recommended HDL Coding Styles**

Altera Corporation

**Send Feedback**
Example 11-2: Verilog HDL Signed Multiplier with Input and Output Registers (Pipelining = 2)

```verilog
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)

```vhdl
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;
END unsigned_mult;
```
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;
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');
            result_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 <= adder_out + pdt_reg + pdt2_reg;
        END IF;
    END process;
    result <= result_reg;
END rtl;
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 PROCESS;
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

<table>
<thead>
<tr>
<th>Language</th>
<th>Full Design Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL</td>
<td>Single-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Single-Port RAM with Initial Contents</td>
</tr>
<tr>
<td></td>
<td>Simple Dual-Port RAM (single clock)</td>
</tr>
<tr>
<td></td>
<td>Simple Dual-Port RAM (dual clock)</td>
</tr>
<tr>
<td></td>
<td>True Dual-Port RAM (single clock)</td>
</tr>
<tr>
<td></td>
<td>True Dual-Port RAM (dual clock)</td>
</tr>
<tr>
<td></td>
<td>Mixed-Width RAM</td>
</tr>
<tr>
<td></td>
<td>Mixed-Width True Dual-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Byte-Enabled Simple Dual-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Byte-Enabled True Dual-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Single-Port ROM</td>
</tr>
<tr>
<td></td>
<td>Dual-Port ROM</td>
</tr>
<tr>
<td>Verilog HDL</td>
<td>Single-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Single-Port RAM with Initial Contents</td>
</tr>
<tr>
<td></td>
<td>Simple Dual-Port RAM (single clock)</td>
</tr>
<tr>
<td></td>
<td>Simple Dual-Port RAM (dual clock)</td>
</tr>
<tr>
<td></td>
<td>True Dual-Port RAM (single clock)</td>
</tr>
<tr>
<td></td>
<td>True Dual-Port RAM (dual clock)</td>
</tr>
<tr>
<td></td>
<td>Single-Port ROM</td>
</tr>
<tr>
<td></td>
<td>Dual-Port ROM</td>
</tr>
<tr>
<td>System Verilog</td>
<td>Mixed-Width Port RAM</td>
</tr>
<tr>
<td></td>
<td>Mixed-Width True Dual-Port RAM</td>
</tr>
<tr>
<td></td>
<td>Mixed-Width True Dual-Port RAM (new data on same port read during write)</td>
</tr>
<tr>
<td></td>
<td>Byte-Enabled Simple Dual Port RAM</td>
</tr>
<tr>
<td></td>
<td>Byte-Enabled True Dual-Port RAM</td>
</tr>
</tbody>
</table>

### 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.
Use Synchronous Memory Blocks

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

```verilog
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

```verilog
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
);```

Recommended HDL Coding Styles
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

link/mwh1409960181641/mwh1409959843979

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` 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 still 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];
```
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

```verilog
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

```vhdl
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
```

Altera Corporation
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

```verilog
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) // we is high
      end
endmodule
```
It is possible to create a single-clock RAM using an assign statement to read the address of \texttt{mem} to create the output \texttt{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**

```verilog
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**

```vhdl
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 (clock);
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**

```verilog
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**

```vhdl
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**

```verilog
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**

```vhdl
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 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)**

```vhdl
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)

```vhdl
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**

```verilog
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**

```verilog
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**

```vhdl
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

```verilog
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

```vhdl
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

```verilog
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

```vhdl
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;
```
BEGIN 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
link/mwh1409960181641/mwh1409959843979

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

```verilog
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

```vhdl
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

```verilog
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
link/mwh1409960181641/mwh1409959843979

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 (\(N \times L \geq 64\)).
- 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 (\(W \times N \times L \geq 32\)).

If the length between each tap (\(L\)) is not a power of two, the software uses more logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or ALMs is required to implement the function. This decode logic eliminates the performance and utilization advantages of implementing shift registers in memory.

The registers that the software maps to the 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**

```verilog
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
            assign sr_out = sr[63];
        end
endmodule
```

**Example 11-36: VHDL Single-Bit Wide, 64-Bit Long Shift Register**

```vhdl
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
                assign sr_out = sr[63];
            END IF;
        END IF;
    END PROCESS;
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

```verilog
data 8xuggp2_d3
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

```vhdl
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;
```

Altera Corporation

Recommended HDL Coding Styles
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.
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**

```verilog
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**

```vhdl
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

[link/mwh1409960181641/mwh1409959843979](link/mwh1409960181641/mwh1409959843979)

**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

```verilog
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)

```vhdl
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 ENTITY dff_control;
```

```vhdl
ARCHITECTURE behav OF dff_control IS
    BEGIN
        PROCESS(clk, aclr, aload)
        BEGIN
            IF (aclr) THEN
                q <= '0';
            ELSIF (aload) THEN
                q <= adata;
            ELSIF (ena) THEN
                q <= data;
            END IF;
        END PROCESS;
```

**Recommended HDL Coding Styles**

Altera Corporation
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**

```vhdl
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**

[link/mwh1409960181641/mwh1409959843979](link/mwh1409960181641/mwh1409959843979)

**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 Stratix devices, the Cyclone series, and MAX II devices, all latches in the **User Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value, such as a D-type input toggling when the enable input is inactive or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is required for a loop around a single LUT. The Quartus 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**

```verilog
module simple_latch (  
    input SetTerm,  
    input ResetTerm,  
    output reg LatchOut  
);  

    // Simple Set-Reset latch
    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.
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.

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**

```verilog
module clock_mux (clk, clk_select, clk_out);
  parameter num_clocks = 4;
  input [num_clocks-1:0] clk;
  input [num_clocks-1:0] clk_select; // one hot
  output clk_out;
  genvar i;
  reg [num_clocks-1:0] ena_r0;
  reg [num_clocks-1:0] ena_r1;
  reg [num_clocks-1:0] ena_r2;
  wire [num_clocks-1:0] qualified_sel;
  // A look-up-table (LUT) can glitch when multiple inputs
  // change simultaneously. Use the keep attribute to
  // insert a hard logic cell buffer and prevent
  // the unrelated clocks from appearing on the same LUT.
  wire [num_clocks-1:0] gated_clks /* synthesis keep */;
  initial begin
    ena_r0 = 0;
    ena_r1 = 0;
    ena_r2 = 0;
  end
  generate
    for (i=0; i<num_clocks; i=i+1)
      begin : lp0
        wire [num_clocks-1:0] tmp_mask;
        assign tmp_mask = {num_clocks{1'b1}} ^ (1 << i);
        assign qualified_sel[i] = clk_select[i] & (~|(ena_r2 &
          tmp_mask));
        always @(posedge clk[i]) begin
          ena_r0[i] <= qualified_sel[i];
          ena_r1[i] <= ena_r0[i];
        end
        always @(negedge clk[i]) begin
          ena_r2[i] <= ena_r1[i];
        end
        assign gated_clks[i] = clk[i] & ena_r2[i];
      end
    endgenerate
  // These will not exhibit simultaneous toggle by construction
  assign clk_out = |gated_clks;
endmodule
```
Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can 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 $\text{sum} = (A + B + C) + (D + E)$ is more likely to create the optimal implementation of a 3-input adder for $A + B + C$ followed by a 3-input adder for $\text{sum1} + D + E$ 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

```verilog
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)
begnin
if (reset)
state <= state_0;
else
state <= next_state;
end
always @ (*)
begnin
    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
din
    assign out = tmp_out_2;
dinendmodule

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
link/mwh1409960181641/mwh1409959843979

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:

  ```verilog
  case (state)
  0: begin
    if (ena) next_state <= state + 2;
    else next_state <= state + 1;
  end
  1: begin
    ...
  endcase
  ```

- No state machine is inferred in the Quartus 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 \texttt{state} 1 and \texttt{state} 2. The temporary variables \texttt{tmp\_out\_0} and \texttt{tmp\_out\_1} store the sum and the difference of \texttt{in\_1} and \texttt{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

```verilog
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
endmodule
```
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;
  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.
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.

**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 11-53: Verilog HDL Binary-Encoded Multiplexers**

```verilog
    case (sel)
        2'b00: z = a;
        2'b01: z = b;
        2'b10: z = c;
        2'b11: z = d;
    endcase
```

**Example 11-54: Verilog HDL One-Hot-Encoded Case Statement**

```verilog
    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**

```vhdl
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.

**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
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:

```vhdl
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):

```vhdl
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.
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):

```< 200
< 2f00
< 1a0
< 100
```

```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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Added information and reference about ramstyle attribute for sif register inference.</td>
</tr>
<tr>
<td>2014.08.18</td>
<td>14.0.a10.0</td>
<td>• Added recommendation to use register pipelining to obtain high performance in DSP designs.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>Removed obsolete MegaWizard Plug-In Manager support.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device support.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Revised section on inserting Altera templates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Code update for Example 11-51.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor corrections and updates.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>• Updated document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor updates and corrections.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Unintentional Latch Generation content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Code update for Example 11-18.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Added support for mixed-width RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated support for no_rw_check for inferring RAM blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for byte-enable</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Updated support for Controlling Inference and Implementation in Device RAM Blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated support for Shift Registers</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Corrected and updated several examples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for Arria II GX devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Other minor changes to chapter</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Updates for the Quartus Prime software version 8.0 release, including:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information to “RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafonctions from HDL Code” on page 6–13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information to “Avoid Unsupported Reset and Control Conditions” on page 6–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information to “Check Read-During-Write Behavior” on page 6–16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added two new examples to “ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafonctions from HDL Code” on page 6–28: Example 6–24 and Example 6–25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added new section: “Clock Multiplexing” on page 6–46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added hyperlinks to references within the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
</tbody>
</table>

**Related Information**

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

### Related Information

- **Migrating to Quartus Prime Pro Edition** on page 1-3
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

<table>
<thead>
<tr>
<th>Compiler Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis &amp; Elaboration</td>
<td>Checks for design file and project errors and builds a one or more design databases. Elaborates all files in the top-level entity hierarchy.</td>
</tr>
<tr>
<td>Analysis &amp; Synthesis</td>
<td>After running Analysis &amp; Elaboration, Analysis &amp; Synthesis performs logic synthesis to optimize, minimize, and technology map design logic to device resources. Analysis &amp; 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).</td>
</tr>
<tr>
<td>Fitter (Place &amp; Route)</td>
<td>After running Analysis &amp; 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.</td>
</tr>
<tr>
<td>Assembler</td>
<td>After running the Fitter, the Assembler converts the Fitter's placement and routing assignments into a programming image for the device.</td>
</tr>
<tr>
<td>TimeQuest Timing Analyzer</td>
<td>After running Analysis &amp; Synthesis or the Fitter, TimeQuest analyzes, debugs, and validates the timing performance of all logic in a design.</td>
</tr>
<tr>
<td>EDA Netlist Writer</td>
<td>After running Analysis &amp; Synthesis or the Fitter, EDA Netlist Writer optionally generates output netlist files for use with other EDA tools.</td>
</tr>
</tbody>
</table>

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
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

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 than 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...
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**

<table>
<thead>
<tr>
<th>Compilation Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyze Current File</td>
<td>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 &gt; Analyze Current File.</td>
</tr>
<tr>
<td>Analysis &amp; Elaboration</td>
<td>Checks your design for syntax and semantic errors and performs elaboration to identify your design hierarchy. To perform Analysis &amp; Elaboration, on the Processing menu, point to Processing &gt; Start &gt; Start Analysis &amp; Elaboration.</td>
</tr>
<tr>
<td>Hierarchy Elaboration</td>
<td>Parses HDL designs and generates a skeleton of hierarchies. Hierarchy Elaboration is similar to the Analysis &amp; Elaboration, but without any elaborated logic, thus making it much faster to generate.</td>
</tr>
<tr>
<td>Analysis &amp; Synthesis</td>
<td>Performs complete Analysis &amp; Synthesis on a design, including technology mapping. To perform Analysis &amp; Synthesis, click Processing &gt; Start &gt; Start Analysis &amp; Synthesis.</td>
</tr>
</tbody>
</table>

**Verilog and SystemVerilog Synthesis Support**

The Analysis & Synthesis Compiler module supports the following Verilog HDL standards:

- Verilog-2001 (IEEE Standard 1364-2001)

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.

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 Types of Clauses for the config_rule_statement Keyword.

Related Information

link/mwh1409960181641/mwh1409959853121

(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:

```verilog
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:

```verilog
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**

```verilog
reg [7:0] ram[0:15];
initial
begin
```

Altera Corporation
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**

- link/mwh1409960181641/mwh1409959856901
- link/mwh1409960181641/mwh1409958382198

**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:


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:

```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**

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 LVTTTL/LVC莫斯 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

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)

<table>
<thead>
<tr>
<th>Available Flows</th>
<th>Flow Sequence</th>
</tr>
</thead>
</table>
| 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)

<table>
<thead>
<tr>
<th>Available Flows</th>
<th>Flow Sequence</th>
</tr>
</thead>
</table>
| 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)

---

**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**

<table>
<thead>
<tr>
<th>VHDL Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.)</td>
</tr>
<tr>
<td>VHDL Version</td>
</tr>
<tr>
<td>- VHDL 1987</td>
</tr>
<tr>
<td>- VHDL 1993</td>
</tr>
<tr>
<td>- VHDL 2008</td>
</tr>
<tr>
<td>Library Mapping File</td>
</tr>
<tr>
<td>File name: [ ]</td>
</tr>
<tr>
<td>[Show information messages describing LMF mapping during compilation]</td>
</tr>
</tbody>
</table>

**Table 12-5: VHDL Input Settings**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL Version</td>
<td>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.</td>
</tr>
<tr>
<td>Library Mapping File</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog Version</td>
<td>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.</td>
</tr>
<tr>
<td>Library Mapping File</td>
<td>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.</td>
</tr>
<tr>
<td>Verilog HDL Macro</td>
<td>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 HDL macro settings list.</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Optimization Modes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced (Normal Flow)</td>
<td>Optimizes synthesis for balanced implementation that respects timing constraints.</td>
</tr>
<tr>
<td>Performance (High effort - increases runtime)</td>
<td>Makes high effort to optimize synthesis for speed performance. High effort increases synthesis run time.</td>
</tr>
<tr>
<td>Performance (Aggressive - increases runtime and area)</td>
<td>Makes aggressive effort to optimize synthesis for speed performance. Aggressive effort increases synthesis run time and device resource use.</td>
</tr>
<tr>
<td>Power (High effort - increases runtime)</td>
<td>Makes high effort to optimize synthesis for low power. High effort increases synthesis run time.</td>
</tr>
<tr>
<td>Area (Aggressive - reduces performance)</td>
<td>Makes aggressive effort to reduce the device area required to implement the design.</td>
</tr>
</tbody>
</table>

Prevent Register Optimizations

The following settings control general register optimization during synthesis.

Table 12-8: Prevent Register Optimizations (Compiler Settings Page)

<table>
<thead>
<tr>
<th>Optimization mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevent register merging</td>
<td>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.</td>
</tr>
<tr>
<td>Prevent register duplication</td>
<td>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.</td>
</tr>
</tbody>
</table>
## Optimization mode

<table>
<thead>
<tr>
<th>Optimization mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevent register retiming</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>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, &quot;ABC&quot; 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 &quot;1001&quot;, because synthesis interprets it as an ASCII string. You must use the type-encoded string B&quot;1001&quot; for synthesis to interpret the parameter value correctly. Altera recommends using the type-encoded format only when necessary to resolve ambiguity.</td>
</tr>
</tbody>
</table>

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

Table 12-9: Synthesis Reports

<table>
<thead>
<tr>
<th>Compiler Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectra-Q Synthesis Summary</td>
<td>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.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis Settings</td>
<td>Lists the value of all synthesis settings during design processing.</td>
</tr>
<tr>
<td>Parallel Compilation</td>
<td>Lists specifications for any use of parallel processing during synthesis.</td>
</tr>
<tr>
<td>Resource Utilization By Entity</td>
<td>Lists the quantity of all types of logic usage for each entity in design synthesis.</td>
</tr>
</tbody>
</table>
### Compiler Module

<table>
<thead>
<tr>
<th>Compiler Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplexor Restructuring Statistics</td>
<td>Provides statistic for the amount of multiplexor restructuring performed during design synthesis.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis IP Cores Summary</td>
<td>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.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis Source Files Read</td>
<td>Lists details about all source files included in design synthesis. Details include file path, file type, and any library information.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis Resource Usage Summary for Partition</td>
<td>Lists the quantity of all types of logic usage for each design partition design synthesis.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis RAM Summary for Partition</td>
<td>Lists RAM usage details for each design partition design synthesis. Details include the name, type, mode, and density.</td>
</tr>
<tr>
<td>Register Statistics</td>
<td>Lists the number of registers using various types of global signals.</td>
</tr>
<tr>
<td>Spectra-Q Synthesis Messages</td>
<td>Lists all info, warning, and error messages that report conditions observed during the Analysis &amp; 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.</td>
</tr>
</tbody>
</table>

### 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.
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.

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

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*
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

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

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.
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.
Reduction 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.
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:

```tcl
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:

```tcl
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

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)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow Any RAM Size for Recognition</td>
<td>Allows the Compiler to infer RAMs of any size, even if they don’t meet the current minimum requirements.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Allow Any ROM Size for Recognition</td>
<td>Allows the Compiler to infer ROMs of any size even if the ROMs do not meet the design's current minimum size requirements.</td>
</tr>
<tr>
<td>Allow Any Shift Register Size for Recognition</td>
<td>Allows the Compiler to infer shift registers of any size even if they do not meet the design's current minimum size requirements.</td>
</tr>
<tr>
<td>Allow Any Shift Register Merging Across Hierarchies</td>
<td>Allows the Compiler to take shift registers from different hierarchies of the design and put them in the same RAM.</td>
</tr>
<tr>
<td>Allow Synchronous Control Signals</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 12-11: Advanced Synthesis Settings (2 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis &amp; Synthesis Message Level</td>
<td>Specifies the type of Analysis &amp; Synthesis messages displayed. <strong>Low</strong> displays only the most important Analysis &amp; Synthesis messages. <strong>Medium</strong> displays most messages, but hides the detailed messages. <strong>High</strong> displays all messages.</td>
</tr>
<tr>
<td>Auto Carry Chains</td>
<td>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 <strong>Carry Chain Length</strong> option. If this option is turned off, CARRY buffers are ignored, but CARRY_SUM buffers are unaffected. The <strong>Auto Carry Chains</strong> option is ignored if you select <strong>Product Term</strong> or <strong>ROM</strong> as the setting for the <strong>Technology Mapper</strong> option.</td>
</tr>
<tr>
<td>Auto Clock Enable Replacement</td>
<td>Allows the Compiler to find logic that feeds a register and move the logic to the register's clock enable input port.</td>
</tr>
<tr>
<td>Auto DSP Block Replacement</td>
<td>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.</td>
</tr>
<tr>
<td>Auto Gated Clock Conversion</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 12-12: Advanced Synthesis Settings (3 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Open-Drain Pins</td>
<td>Allows the Compiler to automatically convert a tri-state buffer with a strong low data input into the equivalent open-drain buffer.</td>
</tr>
<tr>
<td>Auto RAM Replacement</td>
<td>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.</td>
</tr>
<tr>
<td>Auto ROM Replacement</td>
<td>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.</td>
</tr>
<tr>
<td>Auto Resource Sharing</td>
<td>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.</td>
</tr>
<tr>
<td>Auto Shift Register Placement</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Table 12-13: Advanced Synthesis Settings (4 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Design Naming</td>
<td>Specify the naming scheme used for the block design. This option is ignored if it is assigned to anything other than a design entity.</td>
</tr>
<tr>
<td>Carry Chain Length</td>
<td>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.</td>
</tr>
<tr>
<td>Clock MUX Protection</td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable DSP Negate Inferencing</td>
<td>Allows you to specify whether to use the negate port on an inferred DSP block.</td>
</tr>
<tr>
<td>Force Use of Synchronous Clear Signals</td>
<td>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.</td>
</tr>
<tr>
<td>HDL Message Level</td>
<td>Specifies the type of HDL messages you want to view, including messages that display processing errors in the HDL source code. <strong>Level1</strong> displays only the most important HDL messages. <strong>Level2</strong> displays most HDL messages, including warning and information based messages. <strong>Level3</strong> displays all HDL messages, including warning and information based messages and alerts about potential design problems or lint errors.</td>
</tr>
<tr>
<td>Ignore CARRY Buffers</td>
<td>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.</td>
</tr>
<tr>
<td>Ignore CASCADE Buffers</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Table 12-15: Advanced Synthesis Settings (6 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore GLOBAL Buffers</td>
<td>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.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Ignore LCELL Buffers</td>
<td>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.</td>
</tr>
<tr>
<td>Ignore Maximum Fan-Out Assignments</td>
<td>Directs the Compiler to ignore the Maximum Fan-Out Assignments on a node, an entity, or the whole design.</td>
</tr>
<tr>
<td>Ignore ROW GLOBAL Buffers</td>
<td>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.</td>
</tr>
<tr>
<td>Ignore SOFT Buffers</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 12-16: Advanced Synthesis Settings (7 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore Verilog Initial Constructs</td>
<td>Instructs Analysis &amp; Synthesis to ignore initial constructs and variable declaration assignments in your Verilog HDL design files. By default, Analysis &amp; 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.</td>
</tr>
<tr>
<td>Ignore translate_off and synthesis_off Directives</td>
<td>Instructs Analysis &amp; 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.</td>
</tr>
<tr>
<td>Infer RAMS from Raw Logic</td>
<td>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.</td>
</tr>
<tr>
<td>Iteration Limit for Constant Verilog Loops</td>
<td>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.</td>
</tr>
<tr>
<td>Iteration Limit for non-Constant Verilog Loops</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 12-17: Advanced Synthesis Settings (8 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit AHDL integers to 32 Bits</td>
<td>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.</td>
</tr>
<tr>
<td>Maximum DSP Block Usage</td>
<td>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.</td>
</tr>
<tr>
<td>Maximum Number of LABs</td>
<td>Specifies the maximum number of LABs that Analysis &amp; 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.</td>
</tr>
<tr>
<td>Maximum Number of M-RAM/M144K Memory Blocks</td>
<td>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.</td>
</tr>
<tr>
<td>Maximum Number of M4K/M9K/M20K/M10K Memory Blocks</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Table 12-18: Advanced Synthesis Settings (9 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Number of Registers Created from Uninferred RAMs</td>
<td>Specifies the maximum number of registers that Analysis &amp; 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.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NOT Gate Push-Back</td>
<td>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.</td>
</tr>
<tr>
<td>Number of Inverted Registers Reported in Synthesis Report</td>
<td>Specifies the maximum number of inverted registers that the Synthesis Report should display.</td>
</tr>
<tr>
<td>Number of Removed Registers Reported in Synthesis Report</td>
<td>Allows you to specify the maximum number of removed registers that the Synthesis Report should display.</td>
</tr>
<tr>
<td>Optimization Technique</td>
<td>Specifies the overall optimization goal for Analysis &amp; Synthesis: attempt to maximize performance, minimize logic usage, or balance high performance with minimal logic usage.</td>
</tr>
</tbody>
</table>

Table 12-19: Advanced Synthesis Settings (10 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Synthesis</td>
<td>Enables parallel synthesis.</td>
</tr>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>Specifies whether to perform WYSIWYG primitive resynthesis during synthesis. This option uses the setting specified in the Optimization Technique logic option.</td>
</tr>
<tr>
<td>Power-Up Don’t Care</td>
<td>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.</td>
</tr>
<tr>
<td>PowerPlay Power Optimization During Synthesis</td>
<td>Controls the power-driven compilation setting of Analysis &amp; Synthesis. This option determines how aggressively Analysis &amp; 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.</td>
</tr>
</tbody>
</table>
Table 12-20: Advanced Synthesis Settings (11 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Duplicate Registers</td>
<td>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.</td>
</tr>
<tr>
<td>Remove Redundant Logic Cells</td>
<td>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.</td>
</tr>
<tr>
<td>Report Connectivity Checks</td>
<td>Specifies whether the synthesis report should include the panels in the Connectivity Checks folder.</td>
</tr>
<tr>
<td>Report Parameter Settings</td>
<td>Specifies whether the synthesis report should include the panels in the Parameter Settings by Entity Instance folder.</td>
</tr>
<tr>
<td>Report Source Assignments</td>
<td>Specifies whether the synthesis report should include the panels in the Source Assignments folder.</td>
</tr>
</tbody>
</table>

Table 12-21: Advanced Synthesis Settings (12 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Aware Inference for Block RAM</td>
<td>Specifies whether RAM, ROM, and shift-register inference should take the design and device resources into account.</td>
</tr>
<tr>
<td>Restructure Multiplexers</td>
<td>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. <strong>On</strong> minimizes your design area; but may negatively affect design clock speed ($f_{\text{MAX}}$). <strong>Off</strong> disables multiplexer restructuring; it does not decrease logic element usage and does not affect design clock speed ($f_{\text{MAX}}$). You may select <strong>Auto</strong> allows the Quartus Prime software to determine whether multiplexer restructuring should be enabled. The Quartus Prime software uses other synthesis settings, for example, the <strong>Optimization Technique</strong> option, to determine if multiplexer restructuring should be applied to the design; the <strong>Auto</strong> setting will decrease logic element usage but may negatively affect design clock speed ($f_{\text{MAX}}$).</td>
</tr>
<tr>
<td>SDC Constraint Protection</td>
<td>Verifies SDC constraints in register merging. This option helps to maintain the validity of SDC constraints through compilation.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Safe State Machine</td>
<td>Tells the Compiler to implement state machines that can recover from an illegal state.</td>
</tr>
<tr>
<td>Shift Register Replacement</td>
<td>Allows the Compiler to find a group of shift registers of the same length that can be replaced with the alshift_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.</td>
</tr>
</tbody>
</table>

Table 12-22: Advanced Synthesis Settings (13 of 13)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Machine Processing</td>
<td>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.</td>
</tr>
<tr>
<td>Strict RAM Replacement</td>
<td>When this option is On, the Compiler is only allowed to replace RAM if the hardware matches the design exactly.</td>
</tr>
<tr>
<td>Synchronization Register Chain Length</td>
<td>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.</td>
</tr>
<tr>
<td>Synthesis Effort</td>
<td>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.</td>
</tr>
<tr>
<td>Timing-Driven Synthesis</td>
<td>Allows synthesis to use timing information during synthesis to better optimize the design.</td>
</tr>
<tr>
<td>Use LogicLock Constraints during Resource Balancing</td>
<td>Directs the compiler to use LogicLock constraints during DSP and RAM balancing.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALM Register Packing Effort</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Low</strong>—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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Medium</strong>—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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>High</strong>—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.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Auto Global Register</td>
<td>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 <strong>Global Signal</strong> option to <strong>Off</strong> on that signal.</td>
</tr>
<tr>
<td>Control Signals</td>
<td></td>
</tr>
<tr>
<td>Auto Packed Registers</td>
<td>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 <strong>Auto</strong> or <strong>Sparse Auto</strong> 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 <strong>Auto</strong>, 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 <strong>Sparse Auto</strong>, 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 <strong>Off</strong>, the Fitter does not combine registers with other functions. The <strong>Off</strong> setting severely increases the area of the design and may cause a no fit. If this option is set to <strong>Sparse</strong>, the Fitter combines functions in a way which improves performance for many designs. If this option is set to <strong>Normal</strong>, the Fitter combines functions that are expected to maximize design performance and reduce area. When this option is set to <strong>Minimize Area</strong>, 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 <strong>Minimize Area with Chains</strong>, 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 <strong>Off</strong>, registers combine with I/O cells to improve I/O timing (as long as the <strong>Optimize IOC Register Placement For Timing</strong> 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.</td>
</tr>
<tr>
<td>Auto RAM to MLAB Conversion</td>
<td>Specifies whether the Fitter is able to convert RAMs to use LAB locations when those RAMs use <strong>Auto</strong> as the selected block type. If this option is changed to <strong>Off</strong>, then only MLAB cells in the design or RAM cells with a block type setting of <strong>MLAB</strong> use LAB locations to implement memory.</td>
</tr>
<tr>
<td>Auto Register Duplication</td>
<td>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 <strong>Auto Register Duplication</strong> option is also ignored if you select <strong>OFF</strong> as the setting for the <strong>Logic Cell Insertion -- Logic Duplication</strong> logic option. Turning on this option can allow the <strong>Logic Cell Insertion -- Logic Duplication</strong> logic option to improve a design’s routability, but can make formal verification of a design more difficult.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Enable Bus Hold Circuitry</td>
<td>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 <strong>Weak Pull-Up Resistor</strong> option. This option is ignored if it is applied to anything other than a pin.</td>
</tr>
<tr>
<td>Equivalent RAM and MLAB Paused Read Capabilities</td>
<td>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 <strong>Don't Care</strong>, 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 <strong>Care</strong>, 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 <strong>Don't Care</strong>.</td>
</tr>
<tr>
<td>Equivalent RAM and MLAB Power Up</td>
<td>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 <strong>Auto</strong>, 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 <strong>Don't Care</strong>, the same behavior as <strong>Auto</strong> applies, but the message is an information message. If this option is set to <strong>Care</strong>, 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 <strong>Auto</strong> or <strong>Don't Care</strong>.</td>
</tr>
<tr>
<td>Final Placement Optimizations</td>
<td>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 <strong>Automatically</strong> with the <strong>Auto Fit</strong> 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.</td>
</tr>
<tr>
<td>Fitter Aggressive Routability Optimizations</td>
<td>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 <strong>Automatically</strong> setting allows the Fitter decide whether to perform these optimizations based on the routability and timing requirements of the design.</td>
</tr>
</tbody>
</table>
### Table 12-26: Advanced Fitter Settings (4 of 8)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| **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, $f_{\text{MAX}}$, 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.  


<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLAB Add Timing Constraints for mixed-Port Feed-Through Mode Setting Don’t Care</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Table 12-27: Advanced Fitter Settings (5 of 8)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize Design for Metastability</td>
<td>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.</td>
</tr>
<tr>
<td>Optimize Hold Timing</td>
<td>Directs the Fitter to optimize hold time within a device to meet timing requirements and assignments. The following settings are available:</td>
</tr>
<tr>
<td></td>
<td>- I/O Paths and Minimum TPD Paths—Directs the Fitter to meet the following timing requirements and assignments:</td>
</tr>
<tr>
<td></td>
<td>- from I/O pins to registers.</td>
</tr>
<tr>
<td></td>
<td>- from registers to I/O pins.</td>
</tr>
<tr>
<td></td>
<td>- from I/O pins or registers to I/O pins or registers.</td>
</tr>
<tr>
<td></td>
<td>- All Paths—Directs the Fitter to meet the following timing requirements and assignments:</td>
</tr>
<tr>
<td></td>
<td>- $t_H$ from I/O pins to registers.</td>
</tr>
<tr>
<td></td>
<td>- Minimum $t_{CO}$ from registers to I/O pins.</td>
</tr>
<tr>
<td></td>
<td>- Minimum $t_{PD}$ from I/O pins or registers to I/O pins or registers.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Optimize IOC Register Placement for Timing</td>
<td>Specifies whether the Fitter optimizes I/O pin timing by automatically packing registers into I/Os to minimize I/O -&gt; register and register -&gt; 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.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Optimize Multi-Corner Timing</td>
<td>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 <strong>On</strong>, 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 <strong>Off</strong>, 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 <strong>On</strong> 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 <strong>Off</strong> the <strong>Optimize Timing</strong> option, the <strong>Optimize multi-corner timing</strong> option is not available.</td>
</tr>
<tr>
<td>Optimize Timing</td>
<td>Specifies whether the Fitter optimizes to meet the maximum delay timing requirements (for example, clock cycle time). By default, this option is set to <strong>Normal compilation</strong>. Turning it <strong>Off</strong> 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 <strong>Off</strong>, other Fitter timing optimization options have no effect (such as <strong>Optimize Hold Timing</strong>).</td>
</tr>
<tr>
<td>Optimize Timing for ECOs</td>
<td>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 <strong>Off</strong>. Turning it on can improve timing performance at the cost of compilation time.</td>
</tr>
<tr>
<td>Perform Clocking Topology Analysis During Routing</td>
<td>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.</td>
</tr>
<tr>
<td>Periphery to Core Placement and Routing Optimization</td>
<td>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 <strong>Auto</strong>, 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 <strong>On</strong>, the Compiler optimizes all transfers between the periphery and core registers, regardless of timing requirements. Do not setting this option to <strong>On</strong> globally, rather use Assignment Editor to assign optimization to a targeted set of nodes or entities.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Placement Effort Multiplier</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable Power Maximum High-Speed Fraction of Used LAB Tiles</td>
<td>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.</td>
</tr>
<tr>
<td>Programmable Power Technology Optimization</td>
<td>Controls how the Fitter configures tiles to operate in high-speed mode or low-power mode. <strong>Automatic</strong> specifies that the Fitter should minimize power without sacrificing timing performance. <strong>Minimize Power Only</strong> specifies that the Fitter should set the maximum number of tiles to operate in low-power mode. <strong>Force All Used Tiles to High Speed</strong> specifies that the Fitter sets all used tiles to operate in high-speed mode. <strong>Force All Tiles with Failing Timing Paths to High Speed</strong> 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 <strong>Automatic</strong> 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.</td>
</tr>
<tr>
<td>Regenerate Full Fit Reports During ECO Compiles</td>
<td>Controls whether the Fitter report is regenerated during ECO compiles. By default, this option is set to <strong>Off</strong>. Turning it <strong>On</strong> regenerates the report at the cost of compilation time.</td>
</tr>
<tr>
<td>Router Timing Optimization Level</td>
<td>Controls how aggressively the router tries to meet timing requirements. Setting this option to <strong>Maximum</strong> can increase design speed slightly, at the cost of increased compile time. Setting this option to <strong>Minimum</strong> can reduce compile time, at the cost of slightly reduced design speed. The default value is <strong>Normal</strong>.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Spectra-Q Physical Synthesis</td>
<td>Increases circuit performance by performing combinational and sequential optimization during fitting.</td>
</tr>
<tr>
<td>SSN Optimization</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 12-30: Advanced Fitter Settings (8 of 8)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronizer Identification</td>
<td>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.</td>
</tr>
<tr>
<td>Treat Bidirectional Pin as Output Pin</td>
<td>Specifies that the bidirectional pin should be treated as an output pin, meaning that the input path feeds back from the output path.</td>
</tr>
<tr>
<td>Weak Pull-Up Resistor</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Document Revision History

#### Table 12-31: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• First version of document.</td>
</tr>
</tbody>
</table>

Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
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 $t_{HL}$). 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 MTBF 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
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 synchronizer register. Constraining a synchronous input port with a `set_max_delay` constraint for a setup (tSU) 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 tSU and tH of the register. Therefore, these registers are identified as synchronizer 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 synchronizer 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 addition 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:

```tcl
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:

```tcl
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
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 ASYNCHRONOUS"|FORCED|OFF> -to <register or instance name>
```

Synchronizer Data Toggle Rate in MTBF Calculation

To specify a toggle rate for MTBF calculations as described on page Synchronizer Data Toggle Rate in MTBF Calculation, 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>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-append</td>
<td>If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.</td>
</tr>
<tr>
<td>-file &lt;name&gt;</td>
<td>Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type — either *.txt or *.html.</td>
</tr>
<tr>
<td>-panel_name &lt;name&gt;</td>
<td>Sends the results to the panel and specifies the name of the new panel.</td>
</tr>
<tr>
<td>-stdout</td>
<td>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.</td>
</tr>
</tbody>
</table>
**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>
```

**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>
```

**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>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
</tbody>
</table>
### Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated formatting.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Technical edit.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Clarified description of synchronizer identification settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor changes to text and figures throughout document.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
The Quartus Prime software offers several features to enable the detection and correction 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 effect 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.


Related Information

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.
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**
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**

Reconfiguring 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 may require 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.
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. In 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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
</tbody>
</table>
| 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. |
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.
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 VCC and GND.
- Groups pins, nets, wires, module ports, and certain logic into buses where appropriate.
- Displays values in hexadecimal format.
- Converts NOT gates into bubble inversion symbols in the schematic.
- Merges chains of equivalent combinational gates into a single gate; for example, a 2-input AND gate feeding a 2-input AND gate is converted to a single 3-input AND gate.

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 VCC, 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 VCC, 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 to reset the schematic view to encompass the entire design.

Use the **Hand Tool** to change the focus of the viewer 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**
- 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 Viewer 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), and 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

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instances</td>
<td>Modules or instances in the design that can be expanded to lower hierarchy levels.</td>
</tr>
<tr>
<td>Elements</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **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.
To view the properties of an instance or primitive in the RTL Viewer or Technology Map Viewer, right-click the node and click **Properties**.

The **Properties** 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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{CC}$</td>
<td>The port is not connected and has $V_{CC}$ value (tied to $V_{CC}$)</td>
</tr>
<tr>
<td>GND</td>
<td>The port is not connected and has GND value (tied to GND)</td>
</tr>
<tr>
<td>--</td>
<td>The port is connected and has value (other than $V_{CC}$ or GND)</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>Unconnected</td>
<td>The port is not connected and has no value (hanging)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I/O Ports</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="CLK_SEL[1:0]" /></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="RESET_N" /></td>
<td></td>
</tr>
<tr>
<td><strong>I/O Connectors</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="MEM_OE_N [1,3]" /></td>
<td></td>
</tr>
<tr>
<td><strong>OR, AND, XOR Gates</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="always1 always0 C" /></td>
<td></td>
</tr>
<tr>
<td><strong>MULTIPLEXER</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="SEL[2:0] DATA[7:0] Mux5" /></td>
<td></td>
</tr>
<tr>
<td><strong>BUFFER</strong></td>
<td>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.</td>
</tr>
<tr>
<td><img src="image" alt="OE DATAIN OUTO" /></td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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 | 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. |
| Encrypted Instance | A user-defined encrypted instance in the design. The symbol displays the instance name. You cannot open the schematic for the lower-level hierarchy, because the source design is encrypted. |
### Schematic Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>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.</td>
</tr>
<tr>
<td>my_20k_sdp</td>
<td></td>
</tr>
<tr>
<td>CLK0</td>
<td></td>
</tr>
<tr>
<td>CLK1</td>
<td></td>
</tr>
<tr>
<td>CLR0</td>
<td></td>
</tr>
<tr>
<td>PORTAADDRSTALL</td>
<td></td>
</tr>
<tr>
<td>PORTAADDR[8:0]</td>
<td></td>
</tr>
<tr>
<td>PORTABYTEENMASK[3:0]</td>
<td>PORTBDATAOUT[35:0]</td>
</tr>
<tr>
<td>PORTA[35:0]</td>
<td></td>
</tr>
<tr>
<td>PORTAWEN</td>
<td></td>
</tr>
<tr>
<td>PORTWADDRSTALL</td>
<td></td>
</tr>
<tr>
<td>PORTWADDR[8:0]</td>
<td></td>
</tr>
<tr>
<td>PORTB[35:0]</td>
<td></td>
</tr>
</tbody>
</table>

#### Constant

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8’h80</td>
<td>A constant signal value that is highlighted in gray and displayed in hexadecimal format by default throughout the schematic.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
</table>
| A[3:0] Add0 | An adder operator: 
\[
\text{OUT} = A + B
\] |
| B[3:0] | |
| OUT[3:0] | |
| A[0] Mult0 | A multiplier operator: 
\[
\text{OUT} = A \cdot B
\] |
| B[0] | |
| OUT[0] | |
| A[0] Div0 | A divider operator: 
\[
\text{OUT} = A / B
\] |
| B[0] | |
| OUT[0] | |
| A[1:0] Equal3 | Equals |
| B[1:0] | |
| OUT | |
| A[0] ShiftLeft0 | A left shift operator: 
\[
\text{OUT} = (A << \text{COUNT})
\] |
| COUNT[0] | |
| OUT[0] | |
| A[0] ShiftRight0 | A right shift operator: 
\[
\text{OUT} = (A >> \text{COUNT})
\] |
| COUNT[0] | |
| OUT[0] | |
### Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A modulo operator:</td>
</tr>
<tr>
<td></td>
<td>( \text{OUT} = (A \mod B) )</td>
</tr>
<tr>
<td></td>
<td>A less than comparator:</td>
</tr>
<tr>
<td></td>
<td>( \text{OUT} = (A &lt; B \lor A &gt; B) )</td>
</tr>
<tr>
<td></td>
<td>A multiplexer:</td>
</tr>
<tr>
<td></td>
<td>( \text{OUT} = \text{DATA}[\text{SEL}] )</td>
</tr>
<tr>
<td></td>
<td>The data range size is ( 2^{\text{sel range size}} )</td>
</tr>
<tr>
<td></td>
<td>A selector:</td>
</tr>
<tr>
<td></td>
<td>A multiplexer with one-hot select input and more than two input signals</td>
</tr>
<tr>
<td></td>
<td>A binary number decoder:</td>
</tr>
<tr>
<td></td>
<td>( \text{OUT} = (\text{binary_number}(\text{IN}) == x) )</td>
</tr>
<tr>
<td></td>
<td>for ( x = 0 ) to ( x = 2^{(n+1)} - 1 )</td>
</tr>
</tbody>
</table>

**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 display 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 maximum 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.
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.
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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Added Schematic Viewer topic for viewing stage snapshots.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added information for the following new features and feature updates:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Nets visible across hierarchies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Connection Details</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Display Settings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hand Tool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Area Selection Tool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New default behavior for Show/Hide Instance Pins (default is now off)</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>Added Show Netlist on One Page and show/Hide Instance Pins commands.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reorganized and migrated to new template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added support for new Netlist viewer.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>Added sections to support Global Net Routing feature.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Updated screenshots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated chapter for the Quartus Prime software version 10.0, including major user interface changes</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Updated devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor text edits</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Chapter 13 was formerly Chapter 12 in version 8.1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 13–2, Figure 13–3, Figure 13–4, Figure 13–14, and Figure 13–30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added “Enable or Disable the Auto Hierarchy List” on page 13–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Find Command” on page 13–44</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed page size to 8.5” × 11”</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Added Arria GX support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated operator symbols</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated information about the radial menu feature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated zooming feature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated information about probing from schematic to SignalTap II Analyzer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated constant signal information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added .png and .gif to the list of supported image file formats</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated several figures and tables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Renamed several sections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed section “Customizing the Radial Menu”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Moved section “Grouping Combinational Logic into Logic Clouds”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated document content based on the Quartus Prime software version 8.0</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

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

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.

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
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>
<thead>
<tr>
<th>File Extension</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.psp</td>
<td>Precision Synthesis Project File.</td>
</tr>
<tr>
<td>.rep&lt;sup&gt;(10)&lt;/sup&gt;</td>
<td>Synthesis Area and Timing Report File.</td>
</tr>
<tr>
<td>.vqm&lt;sup&gt;(11)&lt;/sup&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

<sup>(10)</sup> The timing report file includes performance estimates that are based on pre-place-and-route information. Use the f<sub>MAX</sub> 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.

<sup>(11)</sup> The Precision Synthesis software-generated VQM file is supported by the Quartus Prime software version 10.1 and later.
<table>
<thead>
<tr>
<th>File Extension</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.tcl</td>
<td>Forward-annotated Tcl assignments and constraints file. The <em>&lt;project name&gt;</em>.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.</td>
</tr>
<tr>
<td>.acf</td>
<td>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.</td>
</tr>
<tr>
<td>.sdc</td>
<td>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 <em>&lt;project name&gt;_pnr_constraints.sdc</em>.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Entry Format for Precision Constraint File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin number</td>
<td><code>set_attribute -name PIN_NUMBER -value &quot;&lt;pin number&gt;&quot; -port &lt;port name&gt;</code></td>
</tr>
<tr>
<td>I/O standard</td>
<td><code>set_attribute -name IOSTANDARD -value &quot;&lt;I/O Standard&gt;&quot; -port &lt;port name&gt;</code></td>
</tr>
<tr>
<td>Drive strength</td>
<td><code>set_attribute -name DRIVE -value &quot;&lt;drive strength in mA&gt;&quot; -port &lt;port name&gt;</code></td>
</tr>
<tr>
<td>Slew rate</td>
<td>`set_attribute -name SLEW -value &quot;TRUE</td>
</tr>
</tbody>
</table>

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:

   ```bash
   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`
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

```plaintext
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

```plaintext
set_max_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

Example 16-7: Using the set_min_delay Constraint

```plaintext
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 represent 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). 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>_vhd` 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 vhdl` 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

```verilog
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;
```
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**

```vhdl
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
        vhd1IP_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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>Use only DSP blocks to implement multipliers, regardless of the size of the multiplier.</td>
</tr>
<tr>
<td>OFF</td>
<td>Use only logic (LUTs) to implement multipliers, regardless of the size of the multiplier.</td>
</tr>
<tr>
<td>AUTO</td>
<td>Use logic (LUTs) or DSP blocks to implement multipliers, depending on the size of the multipliers.</td>
</tr>
</tbody>
</table>

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

```verilog
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 16-13: Setting the dedicated_mult Attribute in VHDL

```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 \( a = b \times c \).

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

```verilog
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 16-15: Setting the `preserve_signal` Attribute in VHDL

```vhdl
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 16-16: Verilog HDL Multiplier Implemented in Logic

```verilog
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

```vhdl
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**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>The ALTMULT_ADD or ALTMULT_ACCUM IP core is inferred.</td>
</tr>
<tr>
<td>FALSE</td>
<td>The ALTMULT_ADD or ALTMULT_ACCUM IP core is not inferred.</td>
</tr>
</tbody>
</table>

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**

```verilog
//synthesis attribute <module name> extract_mac <value>
```

**Example 16-19: Setting the extract_mac Attribute in VHDL**

```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**

```verilog
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**

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
ENTITY signedmult_add IS
    PORT(
        a, b, c, d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
    ATTRIBUTE preserve_signal: BOOLEAN;
    ATTRIBUTE dedicated_mult: STRING;
    ATTRIBUTE extract_mac: BOOLEAN;
    ATTRIBUTE extract_mac OF signedmult_add: ENTITY IS FALSE;
END signedmult_add;
ARCHITECTURE rtl OF signedmult_add IS
    SIGNAL a_int, b_int, c_int, d_int : signed (7 DOWNTO 0);
    SIGNAL pdt_int, pdt2_int : signed (15 DOWNTO 0);
    SIGNAL result_int: signed (15 DOWNTO 0);
    ATTRIBUTE preserve_signal OF pdt_int: SIGNAL IS TRUE;
    ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
    ATTRIBUTE preserve_signal OF pdt2_int: SIGNAL IS TRUE;
    ATTRIBUTE dedicated_mult OF pdt2_int: SIGNAL IS "OFF";
```
BEGIN
    a_int <= signed (a);
    b_int <= signed (b);
    c_int <= signed (c);
    d_int <= signed (d);
    pdt_int <= a_int * b_int;
    pdt2_int <= c_int * d_int;
    result_int <= pdt_int + pdt2_int;
    result <= STD_LOGIC_VECTOR(result_int);
END rtl;

RAM and ROM

The Precision Synthesis software detects memory structures in HDL code and converts them to an operator that infers an ALTSYNCRAM or LPM_RAM_DP 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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Dita conversion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed obsolete devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replaced Megafunction, MegaWizard, and IP Toolbench content with IP Catalog and Parameter Editor content.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.1.1</td>
<td>• Template update.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial changes.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Classic Timing Analyzer support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for <code>.vqm</code> netlist files.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Edited the “Creating Quartus Prime Projects for Multiple EDIF Files” on page 15–30 section for changes with the incremental compilation flow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Editorial changes.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Minor updates for the Quartus Prime software version 10.0 release</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Minor updates for the Quartus Prime software version 9.1 release</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Updated list of supported devices for the Quartus Prime software version 9.0 release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Chapter 11 was previously Chapter 10 in software version 8.1</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>• Changed to 8-1/2 x 11 page size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Title changed to <em>Mentor Graphics Precision Synthesis Support</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated list of supported devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about the Precision RTL Plus incremental synthesis flow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 10-1 to include SystemVerilog</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Guidelines for Altera MegafUNCTIONs and Architecture-Specific Features” on page 10–19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Incremental Compilation and Block-Based Design” on page 10–28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added section “Creating Partitions with the incr_partition Attribute” on page 10–29</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Removed Mercury from the list of supported devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed Precision version to 2007a update 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added note for Stratix IV support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Renamed “Creating a Project and Compiling the Design” section to “Creating and Compiling a Project in the Precision RTL Synthesis Software”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about constraints in the Tcl file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated document based on the Quartus Prime software version 8.0</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
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 (.sv)
- Verilog HDL (.v)
- Functional/RTL Simulation
- Synplify Software
- Constraints & Settings
- Technology-Specific Netlist (.vqm/edf)
- Synopsys Constraints format (.scf) File
- Forward-Annotated Project Constraints (.tcl/.acf)
- Constraints & Settings
- Quartus Prime Software
- No
- Timing & Area Requirements Satisfied?
- Yes
- Post-Place-and-Route Simulation File (.vho/.vo)
- Post-Synthesis Simulation Files (.vho/.vo)
- Gate-Level Timing Simulation
- Gate-Level Functional Simulation
- Configuration/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 (.srn) 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**

<table>
<thead>
<tr>
<th>File Extensions</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.vqm</td>
<td>Technology-specific netlist in .vqm file format. A .vqm file is created for all Altera device families supported by the Quartus Prime software.</td>
</tr>
<tr>
<td>.scf(12)</td>
<td>Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer.</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>File Extension</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.tcl</td>
<td>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.</td>
</tr>
<tr>
<td>.srs</td>
<td>Technology-independent RTL netlist file that can be read only by the Synplify software.</td>
</tr>
<tr>
<td>.srm</td>
<td>Technology view netlist file.</td>
</tr>
<tr>
<td>.acf</td>
<td>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.</td>
</tr>
<tr>
<td>.srr&lt;sup&gt;(13)&lt;/sup&gt;</td>
<td>Synthesis Report file.</td>
</tr>
</tbody>
</table>

### 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 forward-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

<sup>(13)</sup> This report file includes performance estimates that are often based on pre-place-and-route information. Use the $f_{\text{MAX}}$ 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.

<table>
<thead>
<tr>
<th>Relationship Between ( t_{CO} ) and the Output Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{CO} = \text{clock period} - \text{external output delay} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationship Between ( t_{SU} ) and the Input Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{SU} = \text{clock period} - \text{external input delay} )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>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.</td>
</tr>
<tr>
<td>Gray</td>
<td>Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitches.</td>
</tr>
<tr>
<td>One-hot</td>
<td>Generates state machines containing one flipflop for each state. One-hot state machines typically provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than sequential implementations.</td>
</tr>
<tr>
<td>Safe</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Example 17-2: Sample VHDL Code for Applying syn_encoding Directive

```vhdl
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
Altara devices allow register packing into I/O cells. Altara 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 attribute 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**

```tcl
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:

```verilog
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**

```tcl
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**

```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℠). 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>.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**

```verilog
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**

```vhdl
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**

```verilog
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.
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):

```verilog
<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

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn_multstyle</td>
<td>lpm_mult</td>
<td>LPM function inferred and multipliers implemented in DSP blocks.</td>
</tr>
<tr>
<td></td>
<td>logic</td>
<td>LPM function not inferred and multipliers implemented as LEs by the Synplify software.</td>
</tr>
<tr>
<td></td>
<td>block_mult</td>
<td>DSP IP core is inferred and multipliers are mapped directly to DSP block device primitives (for supported devices).</td>
</tr>
</tbody>
</table>

Example 17-10: Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```verilog
module mult(a,b,c,r,en);
  input [7:0] a,b;
  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

```vhdl
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_VECTOR (7 DOWNTO 0);
  SIGNAL mem: Mem_Type;
  SIGNAL addr_reg: STD_LOGIC_VECTOR (6 DOWNTO 0);
BEGIN
  data_out <= mem (CONV_INTEGER(rd_addr));
  PROCESS (clk, we, data_in) BEGIN
    IF (clk='1' AND clk'EVENT) THEN
      IF (we='1') THEN
        mem(CONV_INTEGER(wr_addr)) <= data_in;
      END IF;
    END IF;
  END PROCESS;
END ram_infer;
```
Example 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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Dita conversion. Restructured content.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.1.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Classic Timing Analyzer support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed the “altera_implement_in_esb or altera_Implement_in_eab” section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Edited the “Creating a Quartus Prime Project for Multiple .vqm Files” on page 14–39 section for changes with the incremental compilation flow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Editorial changes.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Minor updates for the Quartus Prime software version 10.0 release.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Minor updates for the Quartus Prime software version 9.1 release.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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 |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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.
Constraints, sometimes known as assignments or logic options, control the way the Quartus® Prime software implements a design for an FPGA. Constraints are also central in the way that the TimeQuest Timing Analyzer and the PowerPlay Power Analyzer inform synthesis, placement, and routing.

There are several types of constraints:

- Global design constraints and software settings, such as device family selection, package type, and pin count.
- Entity-level constraints, such as logic options and placement assignments.
- Instance-level constraints.
- Pin assignments and I/O constraints.

User-created constraints are contained in one of two files: the Quartus Prime Settings File (.qsf) or, in the case of timing constraints, the Synopsys Design Constraints file (.sdc). Constraints and assignments made with the Device dialog box, Settings dialog box, Assignment Editor, Chip Planner, and Pin Planner are contained in the Quartus Prime Settings File. The .qsf file contains project-wide and instance-level assignments for the current revision of the project in Tcl syntax. You can create separate revisions of your project with different settings, and there is a separate .qsf file for each revision.

The TimeQuest Timing Analyzer uses industry-standard Synopsys Design Constraints, also using Tcl syntax, that are contained in Synopsys Design Constraints (.sdc) files. The TimeQuest Timing Analyzer GUI is a tool for making timing constraints and viewing the results of subsequent analysis.

There are several ways to constrain a design, each potentially more appropriate than the others, depending on your tool chain and design flow. You can constrain designs for compilation and analysis in the Quartus Prime software using the GUI, as well as using Tcl syntax and scripting. By combining the Tcl syntax of the .qsf files and the .sdc files with procedural Tcl, you can automate iteration over several different settings, changing constraints and recompiling.

Constraining Designs with the GUI

In the Quartus Prime GUI, the New Project Wizard, Device dialog box, and Settings dialog box allow you to make global constraints and software settings. The Assignment Editor and Pin Planner are spreadsheet-style interfaces for constraining your design at the instance or entity level.

The Assignment Editor and Pin Planner make constraint types and values available based on global design characteristics such as the targeted device. These tools help you verify that your constraints are valid before compilation by allowing you to pick only from valid values for each constraint.
The TimeQuest Timing Analyzer GUI allows you to make timing constraints in SDC format and view the effects of those constraints on the timing in your design. Before running the TimeQuest timing analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and external signal arrival and required times. The Quartus Prime Fitter optimizes the placement of logic in the device to meet your specified constraints.

Global Constraints

Global constraints affect the entire Quartus Prime project and all of the applicable logic in the design. Many of these constraints are simply project settings, such as the targeted device selected for the design. Synthesis optimizations and global timing and power analysis settings can also be applied globally. Global constraints are often made when running the New Project Wizard, or in the Device dialog box or the Settings dialog box, early project development.

Common Types of Global Constraints

The following are the most common types of global constraints:

- Target device specification
- Top-level entity of your design, and the names of the design files included in the project
- Operating temperature limits and conditions
- Physical synthesis optimizations
- Analysis and synthesis options and optimization techniques
- Verilog HDL and VHDL language versions used in your project
- Fitter effort and timing driven compilation settings
- .sdc files for the TimeQuest timing analyzer to use during analysis as part of a full compilation flow

Settings That Direct Compilation and Analysis Flows

Settings that direct compilation and analysis flows in the Quartus Prime software are also stored in the Quartus Prime Settings File for your project, including the following global software settings:

- Settings for EDA tool integration such as third-party synthesis tools, simulation tools, timing analysis tools, and formal verification tools.
- Settings and settings file specifications for the Quartus Prime Assembler, SignalTap II Logic Analyzer, PowerPlay power analyzer, and SSN Analyzer.

Global Constraints and Software Settings

Global constraints and software settings stored in the Quartus Prime settings file are specific to each revision of your design, allowing you to control the operation of the software differently for different revisions. For example, different revisions can specify different operating temperatures and different devices, so that you can compare results.

Only the valid assignments made in the Assignment Editor are saved in the Quartus Prime Settings File, which is located in the project directory. When you make a design constraint, the new assignment is placed on a new line at the end of the file.

When you create or update a constraint in the GUI, the Quartus Prime software displays the equivalent Tcl command in the System tab of the Messages window. You can use the displayed messages as references when making assignments using Tcl commands.
Related Information

Managing Quartus Prime Projects
For more information about how the Quartus Prime software uses Quartus Prime Settings Files

Node, Entity, and Instance-Level Constraints
Node, entity, and instance-level constraints constrain a particular segment of the design hierarchy, as opposed to the entire design. In the Quartus Prime software GUI, most instance-level constraints are made with the Assignment Editor, Pin Planner, and Chip Planner.

Both the Assignment Editor and Pin Planner aid you in correctly constraining your design, both passively, through device-and-assignment-determined pick lists, and actively, through live I/O checking.

You can assign logic functions to physical resources on the device, using location assignments with the Assignment Editor or the Chip Planner. Node, entity, and instance-level constraints take precedence over any global constraints that affect the same sections of the design hierarchy. You can edit and view all node and entity-level constraints you created in the Assignment Editor, or you can filter the assignments by choosing to view assignments only for specific locations, such as DSP blocks.

Constraining Designs with the Pin Planner
The Pin Planner helps you visualize, plan, and assign device I/O pins to ensure compatibility with your PCB layout. The Pin Planner provides a graphical view of the I/O resources in the target device package. You can quickly locate various I/O pins and assign them design elements or other properties.

The Quartus Prime software uses these assignments to place and route your design during device programming. The Pin Planner also helps with early pin planning by allowing you to plan and assign IP interface or user nodes not yet defined in the design.

The Pin Planner Task window provides one-click access to common pin planning tasks. After clicking a pin planning task, you view and highlight the results in the Report window by selecting or deselecting I/O types. You can quickly identify I/O banks, VREF groups, edges, and differential pin pairings to assist you in the pin planning process. You can verify the legality of new and existing pin assignments with the live I/O check feature and view the results in the Live I/O Check Status window.

Constraining Designs with the Chip Planner
The Chip Planner allows you to view the device from a variety of different perspectives, and you can make precise assignments to specific floorplan locations.

With the Chip Planner, you can adjust existing assignments to device resources, such as pins, logic cells, and LABs using drag and drop features and a graphical interface. You can also view equations and routing information, and demote assignments by dragging and dropping assignments to various regions in the Regions window.

Probing Between Components of the Quartus Prime GUI
The Assignment Editor, Chip Planner, and Pin Planner let you locate nodes and instances in the source files for your design in other Quartus Prime viewers.

You can select a cell in the Assignment Editor spreadsheet and locate the corresponding item in another applicable Quartus Prime software window, such as the Chip Planner. To locate an item from the Assignment Editor in another window, right-click the item of interest in the spreadsheet, point to Locate, and click the appropriate command.
You can also locate nodes in the Assignment Editor and other constraint tools from other windows within the Quartus Prime software. First, select the node or nodes in the appropriate window. For example, select an entity in the Entity list in the Hierarchy tab in the Project Navigator, or select nodes in the Chip Planner. Next, right-click the selected object, point to Locate, and click Locate in Assignment Editor. The Assignment Editor opens, or it is brought to the foreground if it is already open.

SDC and the TimeQuest Timing Analyzer

You can make individual timing constraints for individual entities, nodes, and pins with the Constraints menu of the TimeQuest Timing Analyzer. The TimeQuest Timing Analyzer GUI provides easy access to timing constraints, and reporting, without requiring knowledge of SDC syntax.

As you specify commands and options in the GUI, the corresponding SDC or Tcl command appears in the Console. This lets you know exactly what constraint you have added to your Synopsys Design Constraints file, and also enables you to learn SDC syntax for use in scripted flows. The GUI also provides enhanced graphical reporting features.

Individual timing assignments override project-wide requirements. You can also assign timing exceptions to nodes and paths to avoid reporting of incorrect or irrelevant timing violations. The TimeQuest timing analyzer supports point-to-point timing constraints, wildcards to identify specific nodes when making constraints, and assignment groups to make individual constraints to groups of nodes.

Constraining Designs with Tcl

Because .sdc files and .qsf files are both in Tcl syntax, you can modify these files to be part of a scripted constraint and compilation flow.

With Quartus Prime Tcl packages, Tcl scripts can open projects, make the assignments procedurally that would otherwise be specified in a .qsf file, compile a design, and compare compilation results against known goals and benchmarks for the design. Such a script can further automate the iterative process by modifying design constraints and recompiling the design.

Quartus Prime Settings Files and Tcl

QSF files use Tcl syntax, but, unmodified, are not executable scripts. However, you can embed QSF constraints in a scripted iterative compilation flow, where the script that automates compilation and custom results reporting also contains the design constraints.

```tcl
set_global_assignment -name FAMILY "Cyclone II"
set_global_assignment -name DEVICE EP2C35F672C6
set_global_assignment -name TOP_LEVEL_ENTITY chiptrip
set_global_assignment -name ORIGINAL_QUARTUS_VERSION 10.0
set_global_assignment -name PROJECT_CREATION_TIME_DATE "11:45:02  JUNE 08, 2010"
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_instance_assignment -name PARTITION_HIERARCHY root_partition -to | -section_id Top
set_global_assignment -name PARTITION_NETLIST_TYPE SOURCE -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL PLACEMENT_AND_ROUTING | -section_id Top
set_global_assignment -name PARTITION_COLOR 16764057 | -section_id Top
set_global_assignment -name LL_ROOT_REGION ON | -section_id "Root Region"
set_global_assignment -name LL_MEMBER_STATE LOCKED | -section_id "Root Region"
set_global_assignment -name STRATIX_DEVICE_IO_STANDARD "3.3-V LVTTL"
set_location_assignment PIN_P2 -to clk2
```
The example shows the way that the `set_global_assignment` Quartus Prime Tcl command makes all global constraints and software settings, with `set_location_assignment` constraining each I/O node in the design to a physical pin on the device.

However, after you initially create the Quartus Prime Settings File for your design, you can export the contents to a procedural, executable Tcl (.tcl) file. You can then use that generated script to restore certain settings after experimenting with other constraints. You can also use the generated Tcl script to archive your assignments instead of archiving the Quartus Prime Settings file itself.

To export your constraints as an executable Tcl script, on the Project menu, click Generate Tcl File for Project.

```tcl
# Quartus Prime: Generate Tcl File for Project
# File: chiptrip.tcl
# Generated on: Tue Jun 08 13:08:48 2010
# Load Quartus Prime Tcl Project package
package require ::quartus::project
set need_to_close_project 0
set make_assignments 1
# Check that the right project is open
if {[is_project_open]} {
    if {[string compare $quartus(project) "chiptrip"]} {
        puts "Project chiptrip is not open"
        set make_assignments 0
    }
} else {
    # Only open if not already open
    if {[project_exists chiptrip]} {
        project_open -revision chiptrip chiptrip
    } else {
        project_new -revision chiptrip chiptrip
    }
}
```
After setting initial values for variables to control constraint creation and whether or not the project needs to be closed at the end of the script, the generated script checks to see if a project is open. If a project is open but it is not the correct project, in this case, `chiptrip`, the script prints `Project chiptrip is not open` to the console and does nothing else.
If no project is open, the script determines if `chiptrip` exists in the current directory. If the project exists, the script opens the project. If the project does not exist, the script creates a new project and opens the project.

The script then creates the constraints. After creating the constraints, the script writes the constraints to the Quartus Prime Settings File and then closes the project.

### Timing Analysis with Synopsys Design Constraints and Tcl

Timing constraints used in analysis by the Quartus Prime TimeQuest Timing Analyzer are stored in `.sdc` files. Because they use Tcl syntax, the constraints in `.sdc` files can be incorporated into other scripts for iterative timing analysis.

```tcl
# ------------------------------------------
set_time_unit ns
set_decimal_places 3
# ------------------------------------------
# create_clock -period 10.0 -waveform { 0 5.0 } clk2 -name clk2
create_clock -period 4.0 -waveform { 0 2.0 } clk1 -name clk1
# clk1 -> dir*: INPUT_MAX_DELAY = 1 ns
set_input_delay -max 1ns -clock clk1 [get_ports dir*]
# clk2 -> time*: OUTPUT_MAX_DELAY = -2 ns
set_output_delay -max -2ns -clock clk2 [get_ports time*]
```

Similar to the constraints in the Quartus Prime Settings File, you can make the SDC constraints part of an executable timing analysis script.

```tcl
project_open chiptrip
create_timing_netlist
#
# Create Constraints
#
create_clock -period 10.0 -waveform { 0 5.0 } clk2 -name clk2
create_clock -period 4.0 -waveform { 0 2.0 } clk1 -name clk1
# clk1 -> dir*: INPUT_MAX_DELAY = 1 ns
set_input_delay -max 1ns -clock clk1 [get_ports dir*]
# clk2 -> time*: OUTPUT_MAX_DELAY = -2 ns
set_output_delay -max -2ns -clock clk2 [get_ports time*]
#
# Perform timing analysis for several different sets of operating conditions
#
foreach_in_collection oc [get_available_operating_conditions] {
  set_operating_conditions $oc
  update_timing_netlist
  report_timing -setup -npaths 1
  report_timing -hold -npaths 1
  report_timing -recovery -npaths 1
  report_timing -removal -npaths 1
  report_min_pulse_width -nworst 1
} delete_timing_netlist
project_close
```

The script opens the project, creates a timing netlist, then constrains the two clocks in the design and applies input and output delay constraints. The clock settings and delay constraints are identical to those in the `.sdc` file shown in the first example. The next section of the script updates the timing netlist for the constraints and performs multi-corner timing analysis on the design.
A Fully Iterative Scripted Flow

You can use the `::quartus::flow` Tcl package and other packages in the Quartus Prime Tcl API to add flow control to modify constraints and recompile your design in an automated flow. You can combine your timing constraints with the other constraints for your design, and embed them in an executable Tcl script that also iteratively compiles your design as different constraints are applied.

Each time such a modified generated script is run, it can modify the `.qsf` file and `.sdc` file for your project based on the results of iterative compilations, effectively replacing these files for the purposes of archiving and version control using industry-standard source control methods and practices.

This type of scripted flow can include automated compilation of a design, modification of design constraints, and recompilation of the design, based on how you foresee results and pre-determine next-step constraint changes in response to those results.

Related Information

- API Functions for Tcl
- About Quartus Prime Tcl Scripting

Document Revision History

Table 1-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Formatting updates.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>Update Pin Planner description for task and report windows.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Rewrote chapter to more broadly cover all design constraint methods.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed procedural steps and user interface details, and replaced with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>links to Quartus Prime Help.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Added two notes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor text edits.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Revised and reorganized the entire chapter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added section “Probing to Source Design Files and Other Quartus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prime Windows” on page1–2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added description of node type icons (Table1–3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added explanation of wildcard characters.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8½” × 11” page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Updated Quartus Prime software 8.0 revision and date.</td>
</tr>
</tbody>
</table>

Related Information

**Quartus Handbook Archive**
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
This document describes efficient planning and assignment of the I/O pins in your target device. You should consider I/O standards, pin placement rules, and your PCB characteristics early in the design phase.

Figure 2-1: Pin Planner GUI
### Table 2-1: Quartus Prime I/O Pin Planning Tools

<table>
<thead>
<tr>
<th>I/O Planning Task</th>
<th>Click to Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan interfaces and device periphery</td>
<td>Tools &gt; BluePrint Platform Designer</td>
</tr>
<tr>
<td>Edit, validate, or export pin assignments</td>
<td>Assignments &gt; Pin Planner</td>
</tr>
<tr>
<td>Validate pin assignments against design rules</td>
<td>Processing &gt; Start I/O Assignment Analysis</td>
</tr>
</tbody>
</table>

For more information about special pin assignment features for the Arria 10 SoC devices, refer to *Instantiating the HPS Component* in the *Arria 10 Hard Processor System Technical Reference Manual*.

**Related Information**
- BluePrint Planning Overview on page 3-2
- Using the BluePrint Interface on page 3-3
- Instantiating the HPS Component

### I/O Planning Overview

You should plan and assign I/O pins in your design for compatibility with your target device and PCB characteristics. Plan I/O pins early to reduce design iterations and develop an accurate PCB layout sooner. You can assign expected nodes not yet defined in design files, including interface IP core signals, and then generate a top-level file. Specify interfaces for memory, high-speed I/O, device configuration, and debugging tools in your top-level file. The top-level file instantiates the next level of design hierarchy and includes interface port information.

Use the Pin Planner to view, assign, and validate device I/O pin logic and properties. Alternatively, you can enter I/O assignments in a Tcl script, or directly in HDL code. The Pin Planner Task window provides one-click access to I/O planning steps. You can filter and search the nodes in the design. You can define custom groups of pins for assignment. Instantly locate and highlight specific pin types for assignment or evaluation, such as I/O banks, VREF groups, edges, DQ/DQS pins, hard memory interface pins, PCIe hard IP interface pins, hard processor system pins, and clock region input pins. Assign design elements, I/O standards, interface IP, and other properties to the device I/O pins by name or by drag and drop. You can then generate a top-level design file for I/O validation.

Use I/O assignment analysis to fully analyze I/O analysis against VCCIO, VREF, electromigration (current density), Simultaneous Switching Output (SSO), drive strength, I/O standard, PCI_IO clamp diode, and I/O pin direction compatibility rules.

### Basic I/O Planning Flow

The following steps describe the basic flow for assigning and verifying I/O pin assignments:
1. Click **Assignments > Device** and select a target device that meets your logic, performance, and I/O requirements. Consider and specify I/O standards, voltage and power supply requirements, and available I/O pins.

2. Click **Assignments > Pin Planner**.

3. Assign I/O properties to match your device and PCB characteristics, including assigning logic, I/O standards, output loading, slew rate, and current strength.

4. Click **Run I/O Assignment Analysis** in the Tasks pane to validate assignments and generate a synthesized design netlist. Correct any problems reported.

5. Click **Processing > Start Compilation**. During compilation, the Quartus Prime software runs I/O assignment analysis.

### Integrating PCB Design Tools

You can integrate PCB design tools into your workflow to help correctly map pin assignments to the symbols your system circuit schematics and board layout. The Quartus Prime software integrates with board layout tools by allowing import and export of pin assignment information in Quartus Prime Settings Files (.qsf), Pin-Out File (.pin), and FPGA Xchange-Format File (.fx) files. You can integrate PCB tools in the following ways:

#### Table 2-2: Integrating PCB Design Tools

<table>
<thead>
<tr>
<th>PCB Tool Integration</th>
<th>Supported PCB Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define and validate I/O assignments in the Pin Planner, and then export the assignments to the PCB tool for validation</td>
<td>Mentor Graphics® I/O DesignerCadence Allegro</td>
</tr>
<tr>
<td>Define I/O assignments in your PCB tool, and then import the assignments into the Pin Planner for validation</td>
<td>Mentor Graphics® I/O DesignerCadence Allegro</td>
</tr>
</tbody>
</table>
For more information about incorporating PCB design tools, refer to the *Cadence PCB Design Tools Support* and *Mentor Graphics PCB Design Tools Support* chapters in volume 2 of the *Quartus Prime Handbook*.

Related Information

- *Mentor Graphics PCB Design Tools Support* on page 16-1
### Assigning I/O Pins

Use the Pin Planner to visualize, modify, and validate I/O assignments in a graphical representation of the target device. To assign I/O pins, locate the device I/O pin(s) for assignment, enter properties for the pin(s), and validate the legality of the assignment.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Package (BGA example)</td>
<td>Ceramic or plastic heat sink surface mounted with FPGA die and I/O pins or solder balls. In a wire bond BGA example, copper wires connect the bond pads to the solder balls of the package. Click View &gt; Show &gt; Package Top or View &gt; Show &gt; Package Bottom In Pin Planner.</td>
<td><img src="image" alt="Diagram of Device Package" /></td>
</tr>
<tr>
<td>I/O Bank</td>
<td>I/O pins are grouped in I/O banks for assignment of I/O standards. Each numbered bank has its own voltage source pins, called VCCIO pins, for high I/O performance. The specified VCCIO pin voltage is between 1.5 V and 3.3 V. Each bank supports multiple pins with different I/O standards. All pins in a bank must use the same VCCIO signal. Click View &gt; Show &gt; I/O Banks in Pin Planner.</td>
<td><img src="image" alt="Diagram of I/O Bank" /></td>
</tr>
<tr>
<td>I/O Pin</td>
<td>A wire lead or small solder ball on the package bottom or periphery. Each pin has an alphanumeric row and column number. I, O, Q, S, X, and Z are never used. The alphabet is repeated and prefixed with the letter A when exceeded. All I/O pins display by default.</td>
<td><img src="image" alt="Diagram of I/O Pin" /></td>
</tr>
<tr>
<td>Pad</td>
<td>I/O pins are connected to pads located on the perimeter of the top metal layer of the silicon die. Each pad is numbered with an ID starting at 0, and increments by one in a counterclockwise direction around the device. Click View &gt; Pad View in Pin Planner.</td>
<td><img src="image" alt="Diagram of Pad" /></td>
</tr>
<tr>
<td>VREF Pin Group</td>
<td>A group of pins including one dedicated VREF pin required by voltage-referenced I/O standards. A VREF group contains a smaller number of pins than an I/O bank. This maintains the signal integrity of the VREF pin. One or more VREF groups exist in an I/O bank. The pins in a VREF group share the same VCCIO and VREF voltages. Click View &gt; Show &gt; VREF Groups Pin Planner.</td>
<td><img src="image" alt="Diagram of VREF Pin Group" /></td>
</tr>
</tbody>
</table>
Note: You can increase the accuracy of I/O assignment analysis by reserving specific device pins to accommodate undefined but expected I/O.

To assign I/O pins in the Pin Planner, follow these steps:

1. Open a Quartus Prime project, and then click **Assignments > Pin Planner**.
2. Click **Processing > Start Analysis & Elaboration** to elaborate the design and display **All Pins** in the device view.
3. To locate or highlight pins for assignment, click **Pin Finder** or a pin type under **Highlight Pins** in the Tasks pane.
4. (Optional) To define a custom group of nodes for assignment, select one or more nodes in the **Groups** or **All Pins** list, and then click **Create Group**.
5. Enter assignments of logic, I/O standards, interface IP, and properties for device I/O pins in the **All Pins** spreadsheet, or by drag and drop into the package view.
6. To assign properties to differential pin pairs, click **Show Differential Pin Pair Connections**. A red connection line appears between positive (p) and negative (n) differential pins.
7. (Optional) To create board trace model assignments, right-click an output or bidirectional pin, and then click **Board Trace Model**. For differential I/O standards, the board trace model uses a differential pin pair with two symmetrical board trace models. Specify board trace parameters on the positive end of the differential pin pair. The assignment applies to the corresponding value on the negative end of the differential pin pair.
8. To run a full I/O assignment analysis, click **Run I/O Assignment Analysis**. The Fitter reports analysis results. Only reserved pins are analyzed prior to design synthesis.

### Assigning to Exclusive Pin Groups

You can designate groups of pins for exclusive assignment. When you assign pins to an **Exclusive I/O Group**, the Fitter does not place the signals in the same I/O bank with any other exclusive I/O group. For example, if you have a set of signals assigned exclusively to `group_a`, and another set of signals assigned to `group_b`, the Fitter ensures placement of each group in different I/O banks.

### Assigning Slew Rate and Drive Strength

You can designate the device pin slew rate and drive strength. These properties affect the pin’s outgoing signal integrity. Use either the **Slew Rate** or **Slow Slew Rate** assignment to adjust the drive strength of a pin with the **Current Strength** assignment.

**Note:** The slew rate and drive strength apply during I/O assignment analysis.

### Assigning Differential Pins

When you use the Pin Planner to assign a differential I/O standard to a single-ended top-level pin in your design, it automatically recognizes the negative pin as part of the differential pin pair assignment and creates the negative pin for you. The Quartus Prime software writes the location assignment for the negative pin to the `.qsf`; however, the I/O standard assignment is not added to the `.qsf` for the negative pin of the differential pair.

The following example shows a design with `lvds_in` top-level pin, to which you assign a differential I/O standard. The Pin Planner automatically creates the differential pin, `lvds_in(n)` to complete the differential pin pair.
**Note:** If you have a single-ended clock that feeds a PLL, assign the pin only to the positive clock pin of a differential pair in the target device. Single-ended pins that feed a PLL and are assigned to the negative clock pin device cause the design to not fit.

**Figure 2-3: Creating a Differential Pin Pair in the Pin Planner**

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Differential Pair</th>
<th>I/O Standard</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_data[4]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
<tr>
<td>input_data[3]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
<tr>
<td>input_data[2]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
<tr>
<td>input_data[1]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
<tr>
<td>input_data[0]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
<tr>
<td>lvds_in</td>
<td>lvds_in</td>
<td>LVDS</td>
<td>Input</td>
</tr>
<tr>
<td>output_data[7]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[6]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[5]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[4]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[3]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[2]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[1]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>output_data[0]</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Output</td>
</tr>
<tr>
<td>reset</td>
<td></td>
<td>3.3-V LVTTL (default)</td>
<td>Input</td>
</tr>
</tbody>
</table>

If your design contains a large bus that exceeds the pins available in a particular I/O bank, you can use edge location assignments to place the bus. Edge location assignments improve the circuit board routing ability of large buses, because they are close together near an edge. The following shows Altera device package edges.

**Figure 2-4: Die View and Package View of the Four Edges on an Altera Device**
Overriding I/O Placement Rules on Differential Pins

Each device family has predefined I/O placement rules. The I/O placement rules ensure that noisy signals do not corrupt neighboring signals. For example, I/O placement rules define the allowed placement of single-ended I/O with respect to differential pins, or how many output and bidirectional pins can be placed within a VREF group when using voltage referenced input standards. You can use the IO_MAXIMUM_TOGGLE_RATE assignment to override I/O placement rules on pins, such as for system reset pins that do not switch during normal design activity. Setting a value of 0 MHz for this assignment causes the Fitter to recognize the pin at a DC state throughout device operation. The Fitter excludes the assigned pin from placement rule analysis. Do not assign an IO_MAXIMUM_TOGGLE_RATE of 0 MHz to any actively switching pin or your design may not function as intended.

Entering Pin Assignments with Tcl Commands

You can use Tcl scripts to apply pin assignments rather than using the GUI. Enter individual Tcl commands in the Tcl Console, or type the following to apply the assignments contained in a Tcl script:

Example 2-1: Applying Tcl Script Assignments

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

The following example shows use of the `set_location_assignment` and `set_instance_assignment` Tcl commands to assign a pin to a specific location, I/O standard, and drive strength.

Example 2-2: Scripted Pin Assignment

```
set_location_assignment PIN M20 -to address[10]
set_instance_assignment -name IO_STANDARD "2.5 V" -to address[10]
set_instance_assignment -name CURRENT_STRENGTH_NEW "MAXIMUM CURRENT" -to address[10]
```

Related Information

- [Tcl Scripting](#) on page 5-1

Entering Pin Assignments in HDL Code

You can use synthesis attributes or low-level I/O primitives to embed I/O pin assignments directly in your HDL code. When you analyze and synthesize the HDL code, the information is converted into the appropriate I/O pin assignments. You can use either of the following methods to specify pin-related assignments with HDL code:

- Assigning synthesis attributes for signal names that are top-level pins
- Using low-level I/O primitives, such as ALT_BUF_IN, to specify input, output, and differential buffers, and for setting parameters or attributes
Using Low-Level I/O Primitives

You can alternatively enter I/O pin assignments using low-level I/O primitives. You can assign pin locations, I/O standards, drive strengths, slew rates, and on-chip termination (OCT) value assignments. You can also use low-level differential I/O primitives to define both positive and negative pins of a differential pair in the HDL code for your design.

Primitive-based assignments do not appear in the Pin Planner until after you perform a full compilation and back-annotate pin assignments (Assignments > Back Annotate Assignments).

Related Information
Designing with Low Level Primitives User Guide

Importing and Exporting I/O Pin Assignments

The Quartus Prime software supports transfer of I/O pin assignments across projects, or for analysis in third-party PCB tools. You can import or export I/O pin assignments in the following ways:

Table 2-3: Importing and Exporting I/O Pin Assignments

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Import Assignments</th>
<th>Export Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>From your PCB design tool or spreadsheet into Pin Planner during early pin planning or after optimization in PCB tool</td>
<td>From Quartus Prime project for optimization in a PCB design tool</td>
<td></td>
</tr>
<tr>
<td>From another Quartus Prime project with common constraints</td>
<td>From Quartus Prime project for spreadsheet analysis or use in scripting assignments</td>
<td></td>
</tr>
<tr>
<td>From Quartus Prime project for import into another Quartus Prime project with similar constraints</td>
<td>From Quartus Prime project for import into another Quartus Prime project with similar constraints</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Assignments &gt; Import Assignments</th>
<th>Assignments &gt; Export Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>File formats</td>
<td>.qsf, .esf, .acf, .csv, .txt, .sdc</td>
<td>.pin, .fx, .csv, .tcl, .qsf</td>
</tr>
<tr>
<td>Notes</td>
<td>N/A</td>
<td>Exported .csv files retain column and row order and format. Do not modify the row of column headings if importing the .csv file</td>
</tr>
</tbody>
</table>

Importing and Exporting for PCB Tools

The Pin Planner supports import and export of assignments with PCB tools. You can export valid assignments as a .pin file for analysis in other supported PCB tools. You can also import optimized assignment from supported PCB tools. The .pin file contains pin name, number, and detailed properties.

Mentor Graphics I/O Designer requires you to generate and import both an .fx and a .pin file to transfer assignments. However, the Quartus Prime software requires only the .fx to import pin assignments from I/O Designer.
Table 2-4: Contents of .pin File

<table>
<thead>
<tr>
<th>File Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Name/Usage</td>
<td>The name of the design pin, or whether the pin is GND or $V_{CC}$ pin</td>
</tr>
<tr>
<td>Location</td>
<td>The pin number of the location on the device package</td>
</tr>
<tr>
<td>Dir</td>
<td>The direction of the pin</td>
</tr>
<tr>
<td>I/O Standard</td>
<td>The name of the I/O standard to which the pin is configured</td>
</tr>
<tr>
<td>Voltage</td>
<td>The voltage level that is required to be connected to the pin</td>
</tr>
<tr>
<td>I/O Bank</td>
<td>The I/O bank to which the pin belongs</td>
</tr>
<tr>
<td>User Assignment</td>
<td>Y or N indicating if the location assignment for the design pin was user assigned (Y) or assigned by the Fitter (N)</td>
</tr>
</tbody>
</table>

Related Information

Pin-Out Files for Altera Devices

Mentor Graphics PCB Tools Support on page 16-1

Migrating Assignments to Another Target Device

You can migrate compatible pin assignments from one target device to another. You can migrate to a different density and the same device package. You can also migrate between device packages with different densities and pin counts. Click View > Pin Migration Window to verify whether your pin assignments are compatible with migration to a different Altera device.

The Quartus Prime software ignores invalid assignments and generates an error message during compilation. After evaluating migration compatibility, modify any incompatible assignments, and then click Export to export the assignments to another project.
The migration result for the pin function of highlighted PIN_AC23 is not an NC but a voltage reference VREFB1N2 even though the pin is an NC in the migration device. VREF standards have a higher priority than an NC, thus the migration result display the voltage reference. Even if you do not use that pin for a port connection in your design, you must use the VREF standard for I/O standards that require it on the actual board for the migration device.

If one of the migration devices has pins intended for connection to VCC or GND and these same pins are I/O pins on a different device in the migration path, the Quartus Prime software ensures these pins are not used for I/O. Ensure that these pins are connected to the correct PCB plane.

When migrating between two devices in the same package, pins that are not connected to the smaller die may be intended to connect to VCC or GND on the larger die. To facilitate migration, you can connect these pins to VCC or GND in your original design because the pins are not physically connected to the smaller die.

Related Information
AN90: SameFrame PinOut Design for FineLine BGA Packages
Validating Pin Assignments

The Quartus Prime software validates I/O pin assignments against predefined I/O rules for your target device. You can use the following tools to validate your I/O pin assignments throughout the pin planning process:

Table 2-5: I/O Validation Tools

<table>
<thead>
<tr>
<th>I/O Validation Tool</th>
<th>Description</th>
<th>Click to Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Assignment Analysis</td>
<td>Verifies I/O assignment legality of synthesized design against full set of</td>
<td>Processing &gt; Start I/O Assignment Analysis</td>
</tr>
<tr>
<td></td>
<td>I/O rules for the target device</td>
<td></td>
</tr>
<tr>
<td>Advanced I/O Timing</td>
<td>Fully validates I/O assignments against all I/O and timing checks during</td>
<td>Processing &gt; Start Compilation</td>
</tr>
<tr>
<td></td>
<td>compilation</td>
<td></td>
</tr>
</tbody>
</table>

I/O Assignment Validation Rules

I/O Assignment Analysis validates your assignments against the following rules:

Table 2-6: Examples of I/O Rule Checks

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>HDL Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O bank capacity</td>
<td>Checks the number of pins assigned to an I/O bank against the number of pins allowed in the I/O bank.</td>
<td>No</td>
</tr>
<tr>
<td>I/O bank VCCIO voltage capacity</td>
<td>Checks that no more than one VCCIO is required for the pins assigned to the I/O bank.</td>
<td>No</td>
</tr>
<tr>
<td>I/O bank VREF voltage capacity</td>
<td>Checks that no more than one VREF is required for the pins assigned to the I/O bank.</td>
<td>No</td>
</tr>
<tr>
<td>I/O standard and location conflicts</td>
<td>Checks whether the pin location supports the assigned I/O standard.</td>
<td>No</td>
</tr>
<tr>
<td>I/O standard and signal direction conflicts</td>
<td>Checks whether the pin location supports the assigned I/O standard and direction. For example, certain I/O standards on a particular pin location can only support output pins.</td>
<td>No</td>
</tr>
<tr>
<td>Differential I/O standards cannot have open drain turned on</td>
<td>Checks that open drain is turned off for all pins with a differential I/O standard.</td>
<td>No</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>HDL Required?</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>I/O standard and drive strength conflicts</td>
<td>Checks whether the drive strength assignments are within the specifications of the I/O standard.</td>
<td>No</td>
</tr>
<tr>
<td>Drive strength and location conflicts</td>
<td>Checks whether the pin location supports the assigned drive strength.</td>
<td>No</td>
</tr>
<tr>
<td>BUSHOLD and location conflicts</td>
<td>Checks whether the pin location supports BUSHOLD. For example, dedicated clock pins do not support BUSHOLD.</td>
<td>No</td>
</tr>
<tr>
<td>WEAK_PULLUP and location conflicts</td>
<td>Checks whether the pin location supports WEAK_PULLUP (for example, dedicated clock pins do not support WEAK_PULLUP).</td>
<td>No</td>
</tr>
<tr>
<td>Electromigration check</td>
<td>Checks whether combined drive strength of consecutive pads exceeds a certain limit. For example, the total current drive for 10 consecutive pads on a Stratix II device cannot exceed 200 mA.</td>
<td>No</td>
</tr>
<tr>
<td>PCL_IO clamp diode, location, and I/O standard conflicts</td>
<td>Checks whether the pin location along with the I/O standard assigned supports PCL_IO clamp diode.</td>
<td>No</td>
</tr>
<tr>
<td>SERDES and I/O pin location compatibility check</td>
<td>Checks that all pins connected to a SERDES in your design are assigned to dedicated SERDES pin locations.</td>
<td>Yes</td>
</tr>
<tr>
<td>PLL and I/O pin location compatibility check</td>
<td>Checks whether pins connected to a PLL are assigned to the dedicated PLL pin locations.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2-7: Signal Switching Noise Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>HDL Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O bank can not have single-ended I/O when DPA exists</td>
<td>Checks that no single-ended I/O pin exists in the same I/O bank as a DPA.</td>
<td>No</td>
</tr>
<tr>
<td>A PLL I/O bank does not support both a single-ended I/O and a differential signal simultaneously</td>
<td>Checks that there are no single-ended I/O pins present in the PLL I/O Bank when a differential signal exists.</td>
<td>No</td>
</tr>
<tr>
<td>Single-ended output is required to be a certain distance away from a differential I/O pin</td>
<td>Checks whether single-ended output pins are a certain distance away from a differential I/O pin.</td>
<td>No</td>
</tr>
</tbody>
</table>
Running I/O Assignment Analysis

I/O assignment analysis validates I/O assignments against the complete set of I/O system and board layout rules. Full I/O assignment analysis validates blocks that directly feed or are fed by resources such as a PLL, LVDS, or gigabit transceiver blocks. In addition, the checker validates the legality of proper VREF pin use, pin locations, and acceptable mixed I/O standards.

Run I/O assignment analysis during early pin planning to validate initial reserved pin assignments before compilation. Once you define design files, run I/O assignment analysis to perform more thorough legality checks with respect to the synthesized netlist. Run I/O assignment analysis whenever you modify I/O assignments.

The Fitter assigns pins to accommodate your constraints. For example, if you assign an edge location to a group of LVDS pins, the Fitter assigns pin locations for each LVDS pin in the specified edge location and then performs legality checks. To display the Fitter-placed pins, click Show Fitter Placements in the Pin Planner. To accept these suggested pin locations, you must back-annotate your pin assignments.

View the I/O Assignment Warnings report to view and resolve all assignment warnings. For example, a warning that some design pins have undefined drive strength or slew rate. The Fitter recognizes undefined, single-ended output and bidirectional pins as non-calibrated OCT. To resolve the warning, assign the Current Strength, Slew Rate or Slow Slew Rate for the reported pin. Alternatively, you could assign the Termination to the pin. You cannot assign drive strength or slew rate settings when a pin has an OCT assignment.

Running Early I/O Assignment Analysis (without Design Files)

You can perform basic I/O legality checks before defining HDL design files. This technique produces a preliminary board layout. For example, you can specify a target device and enter pin assignments that correspond to PCB characteristics. You can reserve and assign an I/O standards to each pin, and then run I/O assignment analysis to ensure that there are no I/O standard conflicts in each I/O bank.
You must reserve all pins you intend to use as I/O pins, so that the Fitter can determine each pin type. After performing I/O assignment analysis, correct any errors reported by the Fitter and rerun I/O assignment analysis until all errors are corrected. A complete I/O assignment analysis requires all design files.

**Running I/O Assignment Analysis (with Design Files)**

Use I/O assignment analysis to perform full I/O legality checks after fully defining HDL design files. When you run I/O assignment analysis on a complete design, the tool verifies all I/O pin assignments against all I/O rules. When you run I/O assignment analysis on a partial design, the tool checks legality only for defined portions of the design. The following figure shows the workflow for analyzing pin-outs with design files.
Even if I/O assignment analysis passes on incomplete design files, you may still encounter errors during full compilation. For example, you can assign a clock to a user I/O pin instead of assigning it to a dedicated clock pin, or design the clock to drive a PLL that you have not yet instantiated in the design. This occurs because I/O assignment analysis does not account for the logic that the pin drives, and does not verify that only dedicated clock inputs can drive the PLL clock port.

To obtain better coverage, analyze as much of the design as possible over time, especially logic that connects to pins. For example, if your design includes PLLs or LVDS blocks, define these files prior to full analysis. After performing I/O assignment analysis, correct any errors reported by the Fitter and rerun I/O assignment analysis until all errors are corrected.

The following figure shows the compilation time benefit of performing I/O assignment analysis before running a full compilation.
Overriding Default I/O Pin Analysis

You can override the default I/O analysis of various pins to accommodate I/O rule exceptions, such as for analyzing VREF or inactive pins.

Each device contains a number of VREF pins, each supporting a number of I/O pins. A VREF pin and its I/O pins comprise a VREF bank. The VREF pins are typically assigned inputs with VREF I/O standards, such as HSTL- and SSTL-type I/O standards. Conversely, VREF outputs do not require the VREF pin. When a voltage-referenced input is present in a VREF bank, only a certain number of outputs can be present in that VREF bank. I/O assignment analysis treats bidirectional signals controlled by different output enables as independent output enables.

To assign the Output Enable Group option to bidirectional signals to analyze the signals as a single output enable group, follow these steps:

1. To access this assignment in the Pin Planner, right-click the All pins list and click Customize Columns.
2. Under Available columns, add Output Enable Group to Show these columns in this order. The column appears in the All Pins list.
3. Enter the same integer value for the Output Enable Group assignment for all sets of signals that are driving in the same direction.

This assignment is especially important for external memory interfaces. For example, consider a DDR2 interface in a Stratix II device. The device allows 30 pins in a VREF group. Each byte lane for a ×8 DDR2 interface includes one DQS pin and eight DQ pins, for a total of nine pins per byte lane. The DDR2 interface uses the SSTL 18 Class I VREF I/O standard. In typical interfaces, each byte lane has its own output enable. In this example, the DDR2 interface has four byte lanes. Using 30 I/O pins in a VREF group, there are three byte lanes and an extra byte lane that supports the three remaining pins. Without the Output Enable Group assignment, the Fitter analyzes each byte lane as an independent group driven by a unique output enable. In this worst-case scenario the three pins are inputs, and the other 27 pins are outputs violating the 20 output pin limit.
Because DDR2 DQS and DQ pins are always driven in the same direction, the analysis reports an error that is not applicable to your design. The **Output Enable Group** assignment designates the DQS and DQ pins as a single group driven by a common output enable for I/O assignment analysis. When you use the **Output Enable Group** logic option assignment, the DQS and DQ pins are checked as all input pins or all output pins and are not in violation of the I/O rules.

You can also use the **Output Enable Group** assignment to designate pins that are driven only at certain times. For example, the data mask signal in DDR2 interfaces is an output signal, but it is driven only when the DDR2 is writing (bidirectional signals are outputs). To avoid I/O assignment analysis errors, use the **Output Enable Group** logic option assignment to assign the data mask to the same value as the DQ and DQS signals.

You can also use the **Output Enable Group** to designate VREF input pins that are inactive during the time the outputs are driving. This assignment removes the VREF input pins from the VREF analysis. For example, the QVLD signal for an RLDRAM II interface is active only during a read. During a write, the QVLD pin is not active and does not count as an active VREF input pin in the VREF group. Place the QVLD pins in the same output enable group as the RLDRAM II data pins.

**Related Information**

**TimeQuest Timing Analyzer**

**Understanding I/O Analysis Reports**

The detailed I/O assignment analysis reports include the affected pin name and a problem description. The Fitter section of the Compilation report contains information generated during I/O assignment analysis, including the following reports:

- **I/O Assignment Warnings**—lists warnings generated for each pin
- **Resource Section**—quantifies use of various pin types and I/O banks
- **I/O Rules Section**—lists summary, details, and matrix information about the I/O rules tested

The **Status** column indicates whether rules passed, failed, or could not be checked. A severity rating indicates the rule’s importance for effective analysis. “Inapplicable” rules do not apply to the target device family.
Verifying I/O Timing

You must verify board-level signal integrity and I/O timing when assigning I/O pins. High-speed interface operation requires a quality signal and low propagation delay at the far end of the board route. Click **Tools > TimeQuest Timing Analyzer** to confirm timing after making I/O pin assignments. For example, if you change the slew rates or drive strengths of some I/O pins with ECOs, you can verify timing without recompiling the design. You must understand I/O timing and what factors affect I/O timing paths in your design. The accuracy of the output load specification of the output and bidirectional pins affects the I/O timing results.

The Quartus Prime software supports three different methods of I/O timing analysis:

### Table 2-8: I/O Timing Analysis Methods

<table>
<thead>
<tr>
<th>I/O Timing Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced I/O timing analysis</td>
<td>Analyze I/O timing with your board trace model to report accurate, “board-aware” simulation models. Configures a complete board trace model for each I/O standard or pin. TimeQuest applies simulation results of the I/O buffer, package, and board trace model to generate accurate I/O delays and system level signal information. Use this information to improve timing and signal integrity.</td>
</tr>
<tr>
<td>I/O timing analysis</td>
<td>Analyze I/O timing with default or specified capacitive load without signal integrity analysis. TimeQuest reports tCO to an I/O pin using a default or user-specified value for a capacitive load.</td>
</tr>
</tbody>
</table>

---

Verifying I/O Timing

You must verify board-level signal integrity and I/O timing when assigning I/O pins. High-speed interface operation requires a quality signal and low propagation delay at the far end of the board route. Click **Tools > TimeQuest Timing Analyzer** to confirm timing after making I/O pin assignments. For example, if you change the slew rates or drive strengths of some I/O pins with ECOs, you can verify timing without recompiling the design. You must understand I/O timing and what factors affect I/O timing paths in your design. The accuracy of the output load specification of the output and bidirectional pins affects the I/O timing results.

The Quartus Prime software supports three different methods of I/O timing analysis:

### Table 2-8: I/O Timing Analysis Methods

<table>
<thead>
<tr>
<th>I/O Timing Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced I/O timing analysis</td>
<td>Analyze I/O timing with your board trace model to report accurate, “board-aware” simulation models. Configures a complete board trace model for each I/O standard or pin. TimeQuest applies simulation results of the I/O buffer, package, and board trace model to generate accurate I/O delays and system level signal information. Use this information to improve timing and signal integrity.</td>
</tr>
<tr>
<td>I/O timing analysis</td>
<td>Analyze I/O timing with default or specified capacitive load without signal integrity analysis. TimeQuest reports tCO to an I/O pin using a default or user-specified value for a capacitive load.</td>
</tr>
</tbody>
</table>
I/O Timing Analysis | Description
---|---
Full board routing simulation | Use Altera-provided or Quartus Prime software-generated IBIS or HSPICE I/O models for simulation in Mentor Graphics HyperLynx and Synopsys HSPICE.

For more information about advanced I/O timing support, refer to the appropriate device handbook for your target device. For more information about board-level signal integrity and tips on how to improve signal integrity in your high-speed designs, refer to the Altera Signal Integrity Center page of the Altera website.

For information about creating IBIS and HSPICE models with the Quartus Prime software and integrating those models into HyperLynx and HSPICE simulations, refer to the *Signal Integrity Analysis with Third Party Tools* chapter in volume 2 of the *Quartus Prime Handbook*.

Related Information
- Literature and Technical Documentation
- Altera Signal Integrity Center
- *Signal Integrity Analysis with Third-Party Tools* on page 14-1

Running Advanced I/O Timing

Advanced I/O timing analysis uses your board trace model and termination network specification to report accurate output buffer-to-pin timing estimates, FPGA pin and board trace signal integrity and delay values. Advanced I/O timing runs automatically for supported devices during compilation.

Understanding the Board Trace Models

The Quartus Prime software provides board trace model templates for various I/O standards. The following figure shows the template for a 2.5 V I/O standard. This model consists of near-end and far-end board component parameters.

Near-end board trace modeling includes the elements which are close to the device. Far-end modeling includes the elements which are at the receiver end of the link, closer to the receiving device. Board trace model topology is conceptual and does not necessarily match the actual board trace for every component. For example, near-end model parameters can represent device-end discrete termination and breakout traces. Far-end modeling can represent the bulk of the board trace to discrete external memory components, and the far end termination network. You can analyze the same circuit with near-end modeling of the entire board, including memory component termination, and far-end modeling of the actual memory component.
The following figure shows the template for the **LVDS** I/O standard. The far-end capacitance (Cf) represents the external-device or multiple-device capacitive load. If you have multiple devices on the far-end, you must find the equivalent capacitance at the far-end, taking into account all receiver capacitances. The far-end capacitance can be the sum of all the receiver capacitances.

The Quartus Prime software models lossless transmission lines, and does not require a transmission-line resistance value. Only distributed inductance (L) and capacitance (C) values are needed. The distributed L and C values of transmission lines must be entered on a per-inch basis, and can be obtained from the PCB vendor or manufacturer, the CAD Design tool, or a signal integrity tool, such as the Mentor Graphics Hyperlynx software.
Defining the Board Trace Model

The board trace model describes a board trace and termination network as a set of capacitive, resistive, and inductive parameters. Advanced I/O Timing and the SSN Analyzer use the model to simulate the output signal from the output buffer to the far end of the board trace. You can define the capacitive load, any termination components, and trace impedances in the board routing for any output pin or bidirectional pin in output mode. You can configure an overall board trace model for each I/O standard or for specific pins. Define an overall board trace model for each I/O standard in your design. Use that model for all pins that use the I/O standard. You can customize the model for specific pins using the Board Trace Model window in the Pin Planner.

1. Click Assignments > Device and then click Device and Pin Options.
2. Click Board Trace Model and define board trace model values for each I/O standard.
3. Click I/O Timing and define default I/O timing options at board trace near and far ends.
4. Click Assignments > Pin Planner and assign board trace model values to individual pins.

Example 2-3: Specifying Board Trace Model

```bash
# setting the near end series resistance model of sel_p output pin to 25 ohms
```
Related Information

Board Trace Model
For more information about configuring component values for a board trace model, including a complete list of the supported unit prefixes and setting the values with Tcl scripts, refer to Quartus Prime Help.

Modifying the Board Trace Model
To modify the board trace model, click View > Board Trace Model in the Pin Planner. You can modify any of the board trace model parameters within a graphical representation of the board trace model.

The Board Trace Model window displays the routing and components for positive and negative signals in a differential signal pair. Only modify the positive signal of the pair, as the setting automatically applies to the negative signal. Use standard unit prefixes such as p, n, and k to represent pico, nano, and kilo, respectively. Use the short or open value to designate a short or open circuit for a parallel component.

Specifying Near End vs Far End I/O Timing Analysis
You can select a near end or far end point for I/O timing analysis. Near end timing analysis extends to the device pin. You can apply the set_output_delay constraint during near end analysis to account for the delay across the board.

Far end I/O timing analysis, then advanced I/O timing analysis extends to the external device input, at the far end of the board trace. Whether you choose a near end or far end timing endpoint, the board trace models are taken into account during timing analysis.

Understanding Advanced I/O Timing Analysis Reports
View I/O timing analysis information in the following reports:

Table 2-9: Advanced I/O Timing Reports

<table>
<thead>
<tr>
<th>I/O Timing Report</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeQuest Report</td>
<td>Reports signal integrity and board delay data.</td>
</tr>
<tr>
<td>Board Trace Model Assignments report</td>
<td>Summarizes the board trace model component settings for each output and bidirectional signal.</td>
</tr>
<tr>
<td>Signal Integrity Metrics report</td>
<td>Contains all the signal integrity metrics calculated during advanced I/O timing analysis based on the board trace model settings for each output or bidirectional pin. Includes measurements at both the FPGA pin and at the far-end load of board delay, steady state voltages, and rise and fall times.</td>
</tr>
</tbody>
</table>

Note: By default, the TimeQuest analyzer generates the Slow-Corner Signal Integrity Metrics report. To generate a Fast-Corner Signal Integrity Metrics report you must change the delay model by clicking Tools > TimeQuest Timing Analyzer.
Related Information

The TimeQuest Timing Analyzer

Adjusting I/O Timing and Power with Capacitive Loading

When calculating $t_{CO}$ and power for output and bidirectional pins, the TimeQuest analyzer and the PowerPlay Power Analyzer use a bulk capacitive load. You can adjust the value of the capacitive load per I/O standard to obtain more precise $t_{CO}$ and power measurements, reflecting the behavior of the output or bidirectional net on your PCB. The Quartus Prime software ignores capacitive load settings on input pins. You can adjust the capacitive load settings per I/O standard, in picofarads (pF), for your entire design. During compilation, the Compiler measures power and $t_{CO}$ measurements based on your settings. You can also adjust the capacitive load on an individual pin with the Output Pin Load logic option.

Viewing Routing and Timing Delays

Right-click any node and click Locate > Locate in Chip Planner to visualize and adjust I/O timing delays and routing between user I/O pads and $V_{CC}$, GND, and $V_{REF}$ pads. The Chip Planner graphically displays logic placement, LogicLock® Plus regions, relative resource usage, detailed routing information, fan-in and fan-out, register paths, and high-speed transceiver channels. You can view physical timing estimates, routing congestion, and clock regions. Use the Chip Planner to change connections between resources and make post-compilation changes to logic cell and I/O atom placement. When you select items in the Pin Planner, the corresponding item is highlighted in Chip Planner.

Scripting API

You can alternatively use Tcl commands to access I/O management functions, rather than using the GUI. For detailed information about specific scripting command options and Tcl API packages, type the following command at a system command prompt to view the Tcl API Help browser:

```
quartus_sh --qhelp
```

Related Information

- Tcl Scripting on page 5-1
- Command Line Scripting on page 4-1

Generate Mapped Netlist

Enter the following in the Tcl console or in a Tcl script:

```
execute_module -tool map
```

The `execute_module` command is in the flow package.

Type the following at a system command prompt:

```
quartus_syn <project name>
```
Reserve Pins

Use the following Tcl command to reserve a pin:

```
set_instance_assignment -name RESERVE_PIN <value> -to <signal name>
```

Use one of the following valid reserved pin values:

- "AS BIDIRECTIONAL"
- "AS INPUT TRI STATED"
- "AS OUTPUT DRIVING AN UNSPECIFIED SIGNAL"
- "AS OUTPUT DRIVING GROUND"
- "AS SIGNALPROBE OUTPUT"

**Note:** You must include the quotation marks when specifying the reserved pin value.

Set Location

Use the following Tcl command to assign a signal to a pin or device location:

```
set_location_assignment <location> -to <signal name>
```

Valid locations are pin locations, I/O bank locations, or edge locations. Pin locations include pin names, such as `PIN_A3`. I/O bank locations include `IOBANK_1` up to `IOBANK_n`, where `n` is the number of I/O banks in the device.

Use one of the following valid edge location values:

- `EDGE_BOTTOM`
- `EDGE_LEFT`
- `EDGE_TOP`
- `EDGE_RIGHT`

Exclusive I/O Group

Use the following Tcl command to create an exclusive I/O group assignments:

```
set_instance_assignment -name "EXCLUSIVE_IO_GROUP" -to pin
```

Slew Rate and Current Strength

Use the following Tcl commands to create an slew rate and drive strength assignments:

```
set_instance_assignment -name CURRENT_STRENGTH_NEW 8MA -to e[0]
set_instance_assignment -name SLEW_RATE 2 -to e[0]
```

Related Information

*Altera Device Package Information Data Sheet*

Document Revision History

The following table shows the revision history for this chapter.
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2015.11.02      | 15.1.0    | • Removed early pin planning and Live I/O Check support from Quartus Prime Pro Edition handbook  
                    • Changed instances of *Quartus II* to *Quartus Prime*. |
<p>| 2014.12.15      | 14.1.0    | • Updated Live I/O check device support to include only limited device families. |
| 2014.08.30      | 14.0a10.0 | • Added link to information about special pin assignment features for Arria 10 SoC devices. |
| 2014.06.30      | 14.0.0    | • Replaced MegaWizard Plug-In Manager information with IP Catalog.         |
| November 2013   | 13.1.0    | • Reorganization and conversion to DITA.                                  |
| May 2013        | 13.0.0    | • Added information about overriding I/O placement rules.                |
| November 2012   | 12.1.0    | • Updated Pin Planner description for new task and report windows.       |
| June 2012       | 12.0.0    | • Removed survey link.                                                  |
| November 2011   | 11.1.0    | • Minor updates and corrections.                                         |
|                 |           | • Updated the document template.                                         |
| December 2010   | 10.0.1    | Template update                                                          |
| July 2010       | 10.0.0    | • Reorganized and edited the chapter                                     |
|                 |           | • Added links to Help for procedural information previously included in the chapter |
|                 |           | • Added information on rules marked Inapplicable in the I/O Rules Matrix Report |
|                 |           | • Added information on assigning slew rate and drive strength settings to pins to fix I/O assignment warnings |
| November 2009   | 9.1.0     | • Reorganized entire chapter to include links to Help for procedural information previously included in the chapter |
|                 |           | • Added documentation on near-end and far-end advanced I/O timing        |</p>
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>- Updated “Pad View Window” on page 5–20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new figures:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Figure 5–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Figure 5–16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section “Viewing Simultaneous Switching Noise (SSN) Results” on page 5–17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added new section “Creating Exclusive I/O Group Assignments” on page 5–18</td>
</tr>
</tbody>
</table>

Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
Design planning—the feasibility analysis of physical constraints—is a fundamental early step in advanced FPGA design. The BluePrint Platform Designer helps you to accurately plan constraints for physical implementation. Use BluePrint to prototype interface implementations and rapidly define a legal device floorplan. BluePrint supports only Arria 10 devices.

BluePrint interacts dynamically with the Quartus Prime Fitter to accurately verify placement legality while you plan. You can evaluate different floorplans using interactive, real-time reporting to accurately plan the best implementation without iterative compilation. Fitter verification ensures the highest correlation between your BluePrint plan and actual implementation results. You can apply the BluePrint plan constraints to your project with high confidence in the final implementation.

Figure 3-1: BluePrint Platform Designer GUI
BluePrint Platform Designer Functions

- Determines legal design floorplans at any stage of the design process.
- Analyzes clock network scenarios and ensure that specific device regions are fed by high fan-out signals.
- Evaluates placement legality using the Fitter.
- Reports alternative placement to improve performance and resources.
- Supports GUI and Tcl command-line operation.

Related Information

Video Demo: Using BluePrint to Place DDR-3 and PCI Express Gen3

BluePrint Planning Overview

After synthesizing your design, BluePrint helps you to rapidly define a legal device floorplan. BluePrint design planning involves initializing BluePrint, resolving project assignments, placing periphery design elements, and applying the BluePrint plan to your Quartus Prime project.

Note: Implementation constraints that you define in BluePrint do not apply to your Quartus Prime project until you apply the generated BluePrint constraints to your project. Apply the BluePrint constraints to your project by running the generated Tcl script.

Figure 3-2: BluePrint Streamlines Legal Placement

Altera FPGAs contain core and periphery device locations. The device core locations are adaptive look-up tables (ALUTs), core flip-flops, RAMs, and digital signal processors (DSPs). Device periphery locations include I/O elements, phase-locked loops (PLLs), clock buffers, and hard processor systems (HPS).

Altera’s advanced FPGAs contain many silicon features in the periphery, such as hard PCI Express® IP cores, high speed transceivers, hard memory interface circuitry, and embedded processors. Interactions among these periphery atoms can be complex. BluePrint allows you to quickly plan I/O interface and periphery locations for interfaces such as:

- I/O elements
- LVDS interfaces
- PLLs
- Clocks
• Hard interface IP Cores
• High-Speed Transceivers
• Hard Memory Interface IP Cores
• Embedded Processors

After initialization, BluePrint displays your project’s logical hierarchy, post-synthesis design elements, and Fitter-created design elements, alongside a view of target device locations. You can directly place these elements and instantly verify placement in the Fitter to ensure accurate correlation with the final implementation.

**Figure 3-3: BluePrint Planning Flow**

---

**Related Information**

- **Managing Device I/O Pins** on page 2-1
- **Using the BluePrint Interface** on page 3-3

**Using the BluePrint Interface**

The BluePrint user interface helps you to quickly define a plan. You can directly place elements in the graphical floorplan, and verify placement to ensure accurate correlation with the final implementation. Place design elements onto the Chip or Package view of the target device to view real-time, accurate feedback about legal placement options. BluePrint validates your plan against any existing location assignments imported from the project.
Figure 3-4: BluePrint GUI Overview

You can disable or enable imported project assignments to resolve any conflicts and evaluate different implementations. Once you are satisfied with the BluePrint plan for important interfaces, you validate the plan to confirm that the Fitter can place all remaining logic. Once validated, you can apply the BluePrint plan to your project via a generated Tcl file.

Figure 3-5: BluePrint Chip and Package Views

Table 3-1: BluePrint User Interface Reference

<table>
<thead>
<tr>
<th>BluePrint Interface</th>
<th>Planning Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>File tab</td>
<td>Open or close a Quartus Prime project in BluePrint.</td>
</tr>
<tr>
<td>BluePrint Interface</td>
<td>Planning Task</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Home tab</strong></td>
<td>Initialize BluePrint legality engine and load project assignments.</td>
</tr>
<tr>
<td><strong>Assignments tab</strong></td>
<td>Resolve (disable) any conflicting assignments from your synthesized project. Click <strong>Update Plan</strong> to apply your enabled project assignments to the BluePrint plan.</td>
</tr>
<tr>
<td><strong>Plan tab</strong></td>
<td>Interactively plan placement with real-time validation and feedback in a graphical floorplan of the device.</td>
</tr>
<tr>
<td><strong>Reports tab</strong></td>
<td>Generate detailed, interactive reports to inform implementation decisions.</td>
</tr>
<tr>
<td><strong>Tcl Console window</strong></td>
<td>The BluePrint Tcl console displays the underlying Tcl commands executed by the BluePrint user interface. You can also run valid Tcl commands by typing them in the console directly or by sourcing them from a Tcl script.</td>
</tr>
</tbody>
</table>

**Related Information**

- [Step 1: Setup the Project](#) on page 3-5
- [Step 2: Initialize BluePrint](#) on page 3-5
- [Step 3: Update Plan with Project Assignments](#) on page 3-6
- [Step 4: Plan Periphery Placement](#) on page 3-8
- [Step 5: Report Placement Data](#) on page 3-10
- [Step 6: Validate and Apply the Plan](#) on page 3-11

**Step 1: Setup the Project**

BluePrint requires at least a partially completed, synthesized Quartus Prime design project as input. BluePrint also supports planning with a fully complete design project. Before planning in BluePrint, create a Quartus Prime project and set up your initial design:

- Fully define known device periphery interfaces
- Instantiate all known interface IP cores
- Declare all general purpose I/Os
- Define the I/O standard, voltage, drive strength, and slew rate for all general purpose I/Os
- Define the core clocking (optional, but recommended)
- Connect all interfaces of the periphery IP to virtual pins, loopback interference, or test logic to ensure the interfaces are not removed by synthesis

**Step 2: Initialize BluePrint**

To open the current project in BluePrint, click **Tools > BluePrint Platform Designer**. The Quartus Prime software closes and BluePrint opens. Click **Open Project** on the **File** tab to open a different project. Click **Revisions** to open a specific project revision. Click **Home** to return to that tab.

After opening a project in BluePrint, click **Initialize BluePrint** on the **Home** tab to start the Fitter validation and import any project assignments. After initialization, the Fitter runs in the background to dynamically validate your BluePrint plan as you make changes. Changes made in BluePrint are not applied to your Quartus Prime project until you apply the BluePrint plan constraints to your project.
Home Tab Controls

The BluePrint **Home** tab contains the following controls.

**Table 3-2: Home Tab Controls**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize BluePrint</td>
<td>Loads the synthesis netlist, starts the Fitter verification engine, and imports assignments from your Quartus Prime project.</td>
</tr>
<tr>
<td>Reset Plan</td>
<td>Unplaces all placed design elements and removes applied project assignments from the BluePrint plan. Resets all project assignments to the enabled state. You must subsequently run <strong>Update Plan</strong> prior to placing design elements. This command only applies to your BluePrint plan and does not impact your Quartus Prime project assignments until you apply the BluePrint script.</td>
</tr>
</tbody>
</table>

**Step 3: Update Plan with Project Assignments**

Before periphery planning you must update the BluePrint plan with the imported project assignments. You must resolve any conflicting project assignments before clicking **Update Plan** to apply the enabled project assignments to your BluePrint plan.

Click the **Assignments** tab to review and reconcile any conflicting assignments that BluePrint imports from your project. Enable or disable specific project assignments to resolve any conflicts. After resolving all conflicts, click **Update Plan** to apply the enabled project assignments to your BluePrint plan. You can click **Reset Plan** to reset all project assignments to the enabled state.
Assignments Tab Controls

The BluePrint Assignments tab contains the following controls for resolving any potential conflicts with imported project assignments. Enable or disable specific or classes of assignments until you resolve all potential conflicts. After you are satisfied with the status of all project assignments, click Update Plan to update your BluePrint plan with the enabled project assignments. BluePrint reports an error for any remaining assignment conflicts.

Table 3-3: Assignments Tab Controls

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter field</td>
<td>Supports creation of wildcard expressions for assignment targets.</td>
</tr>
<tr>
<td>Enable All Project Assignments</td>
<td>Enables all imported project assignments for application to your BluePrint plan.</td>
</tr>
<tr>
<td>Disable All Project Assignments</td>
<td>Disables all imported project assignments in the plan.</td>
</tr>
<tr>
<td>Update Plan</td>
<td>Applies the enabled project assignments to your BluePrint plan. You cannot perform periphery planning on the Plan tab until you update the plan.</td>
</tr>
<tr>
<td>Enabled/Disabled</td>
<td>Filters the list to show only enabled or disabled assignments in the BluePrint plan.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clears any filter from the Assignments list.</td>
</tr>
<tr>
<td>Copy Current View</td>
<td>Copies the current view to the clipboard.</td>
</tr>
</tbody>
</table>
Quartus Prime | Closes BluePrint and opens the Quartus Prime software, allowing you to change and resynthesize your project assignments before planning with Blueprint. The Quartus Prime software automatically closes when you open BluePrint.

**Step 4: Plan Periphery Placement**

Click the Plan tab to interactively place IP cores and other design elements in legal locations in the device periphery. The Plan tab displays a list of your project’s design elements, alongside a graphical abstraction of the target device architecture.

**Figure 3-8: BluePrint Platform Designer (Plan Tab)**

You can assign design elements by dragging them from the **Design Elements** list and dropping them onto available device resources in the Chip or Package view. Alternatively, click the button next to any design element to display a list of legal locations. Single-click any legal location in the list to highlight the location in the floorplan. Double-click any legal location in the list to place in the floorplan. BluePrint displays the approximate number of legal locations in the Legal Locations column. Clicking the button verifies the number of legal locations.
You can apply various filters to locate specific design elements or target device resources. Device resources are color coded by type, allowing quick access to any structure. BluePrint accurately evaluates the constraints using the Fitter in real time. Place elements in the following order for efficiency:

1. All I/O pins or elements, such as PLLs, with known specific location requirements
2. All known periphery interface IP
3. (Optional) Remaining unplaced cells (Autoplace Fixed or right-click any individual unplaced element to Autoplace Selected Element).

Select any physical design element and click **Report Placeability of Selected Element** to generate a report listing all legal placement options on the **Reports** tab. You can **Export** the contents of the report for analysis.

**Note:** Changes made in BluePrint do not apply to your Quartus Prime project until you apply the generated BluePrint plan constraints script to your project.

### Plan Tab Controls

The BluePrint **Plan** tab contains the following controls to help you place logic in the BluePrint plan. Placement or unplacement in the BluePrint plan does not apply to your Quartus Prime project until you add the generated BluePrint constraints script to your project.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Click to list legal locations" /></td>
<td>Click to list legal locations.</td>
</tr>
<tr>
<td><strong>Autoplace All</strong></td>
<td>Attempts to place all unplaced design elements in legal locations in the BluePrint plan.</td>
</tr>
<tr>
<td><strong>Autoplace Fixed</strong></td>
<td>Attempts to place all unplaced design elements that have only one legal location into the BluePrint plan.</td>
</tr>
<tr>
<td><strong>Unplace All</strong></td>
<td>Unplaces all placed design elements in the BluePrint plan.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Validate plan</td>
<td>Verifies that all constraints in the BluePrint plan are compatible with placement of all remaining unplaced design elements. You can then directly locate and resolve the source of any reported validation errors. You must successfully validate the plan before running <strong>Write Plan File</strong>.</td>
</tr>
<tr>
<td>Write Plan File</td>
<td>Saves your BluePrint plan to a file for subsequent application to your project. This command is available only after successfully running <strong>Validate Plan</strong>.</td>
</tr>
<tr>
<td>Right-click &gt; Auto-place selected element</td>
<td>Attempts to place the selected design element and all of its children in a legal location in the BluePrint plan.</td>
</tr>
<tr>
<td>Chip View</td>
<td>Displays the target device chip. Zoom in to display chip details.</td>
</tr>
<tr>
<td>Package View</td>
<td>Displays the target device package. Zoom in to display chip details.</td>
</tr>
<tr>
<td>Right-click &gt; Report Placeability of Selected Element</td>
<td>Displays detailed information on the <strong>Reports</strong> tab, showing legal locations in the BluePrint plan for the selected cell in order of suitability for fitting.</td>
</tr>
<tr>
<td>Copy Current View</td>
<td>Copies the current BluePrint plan to the clipboard for pasting into other files, such as word processing or presentation files.</td>
</tr>
</tbody>
</table>

**Step 5: Report Placement Data**

BluePrint provides detailed, actionable reporting to help you quickly implement the best plan for your design. Generate BluePrint placement and connectivity reports to help locate cells and make the best decisions about placement for the interfaces and elements in your design. You can access reporting commands from the **Reports** or **Plan** tab after you initialize (**Initialize BluePrint**) and apply enabled project assignments (**Update Plan**) in BluePrint.

Select any element in the **Plan** tab to **Report Placeability of Selected Element**, or **Report Legal Locations of Selected Element**. The generated reports appear on the BluePrint **Reports** tab. The BluePrint reports are dynamic, allowing you to select elements in the report and place, unplace, or report detailed data about the selected element(s) or location(s). This document describes each report in detail.
Step 6: Validate and Apply the Plan

You must validate your BluePrint plan before applying the BluePrint plan constraints to your project using a generated Tcl script. Validation confirms that the Fitter can place all remaining unplaced design elements. When you are satisfied with your BluePrint plan, follow these steps to validate and apply the BluePrint plan to your Quartus Prime project:

1. Click **Validate Plan** to confirm that the Fitter can place all remaining unplaced design elements.
2. After validation, click **Write Plan File** to generate a Tcl script that applies the plan to your project. The output Tcl file contains instructions to apply the BluePrint plan to your Quartus Prime project. Close BluePrint and open the Quartus Prime software.
3. Type the following from a command prompt to run the Tcl script and apply the BluePrint plan to your Quartus Prime project:

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

   Alternatively, you can source the script from the Quartus Prime GUI.
4. In the Quartus Prime software, click **Start > Start Analysis & Synthesis** to synthesis and apply the BluePrint plan in your project.

BluePrint User Interface Controls

The BluePrint user interface includes the following controls for planning your design platform.
Home Tab Controls

The BluePrint Home tab contains the following controls.

Table 3-5: Home Tab Controls

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize BluePrint</td>
<td>Loads the synthesis netlist, starts the Fitter verification engine, and imports assignments from your Quartus Prime project.</td>
</tr>
<tr>
<td>Reset Plan</td>
<td>Unplaces all placed design elements and removes applied project assignments from the BluePrint plan. Resets all project assignments to the enabled state. You must subsequently run Update Plan prior to placing design elements. This command only applies to your BluePrint plan and does not impact your Quartus Prime project assignments until you apply the BluePrint script.</td>
</tr>
</tbody>
</table>

Assignments Tab Controls

The BluePrint Assignments tab contains the following controls for resolving any potential conflicts with imported project assignments. Enable or disable specific or classes of assignments until you resolve all potential conflicts. After you are satisfied with the status of all project assignments, click Update Plan to update your BluePrint plan with the enabled project assignments. BluePrint reports an error for any remaining assignment conflicts.

Table 3-6: Assignments Tab Controls

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter field</td>
<td>Supports creation of wildcard expressions for assignment targets.</td>
</tr>
<tr>
<td>Enable All Project Assignments</td>
<td>Enables all imported project assignments for application to your BluePrint plan.</td>
</tr>
<tr>
<td>Disable All Project Assignments</td>
<td>Disables all imported project assignments in the plan.</td>
</tr>
<tr>
<td>Update Plan</td>
<td>Applies the enabled project assignments to your BluePrint plan. You cannot perform periphery planning on the Plan tab until you update the plan.</td>
</tr>
<tr>
<td>Enabled/Disabled</td>
<td>Filters the list to show only enabled or disabled assignments in the BluePrint plan.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clears any filter from the Assignments list.</td>
</tr>
<tr>
<td>Copy Current View</td>
<td>Copies the current view to the clipboard.</td>
</tr>
<tr>
<td>Quartus Prime</td>
<td>Closes BluePrint and opens the Quartus Prime software, allowing you to change and resynthesize your project assignments before planning with BluePrint. The Quartus Prime software automatically closes when you open BluePrint.</td>
</tr>
</tbody>
</table>
Plan Tab Controls

The BluePrint Plan tab contains the following controls to help you place logic in the BluePrint plan. Placement or unplacement in the BluePrint plan does not apply to your Quartus Prime project until you add the generated BluePrint constraints script to your project.

Table 3-7: Plan Tab Controls

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click to list legal locations.</td>
<td></td>
</tr>
<tr>
<td>Autoplace All</td>
<td>Attempts to place all unplaced design elements in legal locations in the BluePrint plan.</td>
</tr>
<tr>
<td>Autoplace Fixed</td>
<td>Attempts to place all unplaced design elements that have only one legal location into the BluePrint plan.</td>
</tr>
<tr>
<td>Unplace All</td>
<td>Unplaces all placed design elements in the BluePrint plan.</td>
</tr>
<tr>
<td>Validate plan</td>
<td>Verifies that all constraints in the BluePrint plan are compatible with placement of all remaining unplaced design elements. You can then directly locate and resolve the source of any reported validation errors. You must successfully validate the plan before running Write Plan File.</td>
</tr>
<tr>
<td>Write Plan File</td>
<td>Saves your BluePrint plan to a file for subsequent application to your project. This command is available only after successfully running Validate Plan.</td>
</tr>
<tr>
<td>Right-click &gt; Auto-place selected element</td>
<td>Attempts to place the selected design element and all of its children in a legal location in the BluePrint plan.</td>
</tr>
<tr>
<td>Chip View</td>
<td>Displays the target device chip. Zoom in to display chip details.</td>
</tr>
<tr>
<td>Package View</td>
<td>Displays the target device package. Zoom in to display chip details.</td>
</tr>
<tr>
<td>Right-click &gt; Report Placeability of Selected Element</td>
<td>Displays detailed information on the Reports tab, showing legal locations in the BluePrint plan for the selected cell in order of suitability for fitting.</td>
</tr>
<tr>
<td>Copy Current View</td>
<td>Copies the current BluePrint plan to the clipboard for pasting into other files, such as word processing or presentation files.</td>
</tr>
</tbody>
</table>

Reports Tab Controls

The BluePrint Reports tab contains the following controls to help you filter data and find entities and locations.

Table 3-8: Reports Tab Controls

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create all Summary Reports</td>
<td>Creates the following summary reports:</td>
</tr>
<tr>
<td></td>
<td>• BluePrint Platform Designer Summary</td>
</tr>
<tr>
<td></td>
<td>• All Periphery Cells</td>
</tr>
<tr>
<td></td>
<td>• Placed/Unplaced Periphery Cells</td>
</tr>
<tr>
<td></td>
<td>• Periphery Location Types</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Report All Placed/Unplaced Pins</strong></td>
<td>Reports the name, parent (if any), and type of all placed (Report All Placed Pins) or unplaced (Report All Unplaced Pins) pins in the BluePrint plan, respectively. The Placed Pins report includes the placement location name. The Unplaced Pins report includes the number of potential placement locations. Right-click any cell to place, unplace, or report connectivity or location information.</td>
</tr>
<tr>
<td><strong>Report All Placed/Unplaced Channels</strong></td>
<td>Reports the name, parent (if any), and type of all placed (Report All Placed HSSI Channels) or unplaced (Report All Unplaced HSSI Channels) channels in the BluePrint plan, respectively. The Placed HSSI Channels report includes the placement location name. The Unplaced HSSI Channels report includes the number of potential placement locations. Right-click any cell to place, unplace, or report connectivity or location information.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Placed/Unplaced Periphery Cells of Selected Type</strong></td>
<td>Reports the name, parent (if any), and type of placed (Report Placed Periphery Cells of Selected Type) or unplaced (Report Unplaced Periphery Cells of Selected Type) cells matching the selected type. The placed cells report includes the placement location name. The unplaced cells report includes the number of potential placement locations. Right-click any cell to place, unplace, or report connectivity or location information.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Periphery Locations of Selected Type</strong></td>
<td>Reports all locations in the device of the selected type, and whether the location supports merging.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Periphery Cell Connectivity</strong></td>
<td>Reports the source port and type, destination port and type, of connections to the selected cell. Right-click any cell to report the individual cell connectivity.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Place/Unplace Cell</strong></td>
<td>Places the cell in the selected location of the BluePrint plan. Similarly, you can right-click any cell and then click Place Cell of Selected Type or Unplace Cell of Selected Type to place or unplace multiple cells of the same type.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Cell Locations for Custom Placement</strong></td>
<td>Reports the preferred legal locations for the selected cell in the BluePrint plan in the Legal Location report. Right-click to immediately place the cell in a location or report all periphery location of the selected type.</td>
</tr>
<tr>
<td><strong>Remove Invalid Reports</strong></td>
<td>Removes outdated BluePrint reports that you invalidate by changes to the BluePrint plan.</td>
</tr>
<tr>
<td><strong>Copy Current View</strong></td>
<td>Copies the current report view to the clipboard for pasting into other applications.</td>
</tr>
</tbody>
</table>

**BluePrint Reports**

BluePrint provides detailed, actionable feedback to help you quickly implement the best plan for your design. You can access placement and further reporting functions directly from BluePrint reports. Use the reports to help locate cells and assign suitable placement locations for specific interfaces and elements in your design. BluePrint generates the following reports that provide detailed planning information.
Report Summary

Click the Create all Summary Reports on the Reports tab to generate summary reports about periphery cells in the BluePrint plan. Right-click any cell type to report placed, unplaced, connectivity, or location information.

Figure 3-11: Summary Reports

Table 3-9: Report Summary

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create all Summary Reports</td>
<td>Creates the following summary reports:</td>
</tr>
<tr>
<td></td>
<td>- BluePrint Platform Designer Summary—reports software version and total number of periphery and top-level periphery cells.</td>
</tr>
<tr>
<td></td>
<td>- All Periphery Cells— reports the name, parent, and type of all periphery cells in the design.</td>
</tr>
<tr>
<td></td>
<td>- Placed/Unplaced Periphery Cells—reports the name, parent, and type of all placed and unplaced periphery cells in the BluePrint plan.</td>
</tr>
<tr>
<td></td>
<td>- Periphery Location Types—reports the number of each type of periphery location available in the target device and the number required by your design.</td>
</tr>
</tbody>
</table>

Report Pins

Click the following Reports tab commands to generate reports about I/O pins in the design. Right-click any cell type to place, unplace, or report connectivity or location information.
### Table 3-10: Report Pin Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Report All Placed Pins</strong></td>
<td>Generates the Placed Pins report. This report lists the name, parent, type, and location of all placed pins in the BluePrint plan.</td>
</tr>
<tr>
<td><strong>Report All Unplaced Pins</strong></td>
<td>Generates the Unplaced Pins report. This report lists the name, parent, type, and the number of potential placements for all unplaced pins in the BluePrint plan.</td>
</tr>
</tbody>
</table>

**Figure 3-12: Placed Pins Report**

![Placed Pins Report](image)

**Figure 3-13: Unplaced Pins Report**

![Unplaced Pins Report](image)
Report HSSI Channels

Click the following Reports tab commands to generate reports about HSSI channels in the BluePrint plan. Right-click any cell type to place, unplace, or report connectivity or location information.

Table 3-11: Report Channel Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report All Placed HSSI Channels</td>
<td>Generates the Placed HSSI Channels report. This report lists the name, parent, type, and location of all placed HSSI RX/TX channels in the BluePrint plan.</td>
</tr>
<tr>
<td>Report All Unplaced HSSI Channels</td>
<td>Generates the Unplaced HSSI Channels report. This report lists the name, parent, type, and location of all unplaced HSSI RX/TX channels in the BluePrint plan.</td>
</tr>
</tbody>
</table>

Figure 3-14: Unplaced HSSI Channels Report

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Parent Name</th>
<th>Cell Type</th>
<th>Potential Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI RX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI receive channel</td>
<td>1L</td>
<td></td>
</tr>
<tr>
<td>HSSI TX CHANNEL CluserHSSI DUPLEX CHANNEL CLUS</td>
<td>HSSI transmit channel</td>
<td>1L</td>
<td></td>
</tr>
</tbody>
</table>

Report Clocks

Click the following Reports tab commands to generate reports showing clock networks in the plan. Use this report to analyze clock network scenarios and ensure that specific device regions are fed by high fan-out signals.

Table 3-12: Report Clocks Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Clocks</td>
<td>Generates the Global and other Fast Signals report.</td>
</tr>
</tbody>
</table>
Report Legal Locations

In the **Plan** tab, right-click on a Design Element to generate reports that show legal locations for the element. In the **Reports** tab, select the Legal Locations report for the element. Right-click any Device Location Name to place the cell. Right-click any cell to immediately place the cell in the selected locations or report all periphery location of the selected type.

**Table 3-13: Report Legal Locations Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click &gt; Report Legal Locations of Selected Element</td>
<td>Accessible in the <strong>Plan</strong> tab, this command generates the Legal Locations report that shows legal locations for the selected cell in order of suitability for fitting.</td>
</tr>
<tr>
<td>Right-click &gt; Report Cell Locations for Custom Placement</td>
<td>Accessible in any of the &quot;Unplaced&quot; reports, this command generates the Legal Locations report that shows legal locations for your custom placement.</td>
</tr>
</tbody>
</table>

**Figure 3-16: Legal Locations Report**
Report Periphery Locations

Click the following Reports tab command to generate reports that show the status of periphery cells in the BluePrint plan.

Table 3-14: Report Periphery Locations Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Right-click &gt; Report Placed Periphery Cells of Selected Type</strong></td>
<td>Accessible from the All Periphery Cells report. This command reports the name, parent (if any), type, and location of the selected placed periphery cells matching the selected type. Right-click any cell to place, unplace, or report connectivity or location information.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Unplaced Periphery Cells of Selected Type</strong></td>
<td>Accessible from the All Periphery Cells report. This command reports the name, parent (if any), type, and number of suitable locations for the selected unplaced periphery cells matching the selected type. Right-click any cell to place, unplace, or report connectivity or location information.</td>
</tr>
<tr>
<td><strong>Right-click &gt; Report Periphery Locations of Selected Type</strong></td>
<td>Reports all locations in the device of the selected type, and whether the location supports merging.</td>
</tr>
</tbody>
</table>

Figure 3-17: Periphery Locations Report

<table>
<thead>
<tr>
<th>Placed Periphery Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists placed cells, their parent cell if applicable, their type, and where they are placed on the device</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Parent Name</th>
<th>Cell Type</th>
<th>Location Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ALTEGRA CLKUSR~--CLUSTER</td>
<td>-ALTEGRA CLKUSR~--CLUSTER</td>
<td>I/O pin cluster</td>
<td>BD32</td>
</tr>
<tr>
<td>-ALTEGRA CLKUSR~</td>
<td>-ALTEGRA CLKUSR~--CLUSTER</td>
<td>I/O pad</td>
<td>PIN_BD32</td>
</tr>
<tr>
<td>-ALTEGRA DATA0~--CLUSTER</td>
<td>-ALTEGRA CLKUSR~--CLUSTER</td>
<td>I/O pin buffer</td>
<td>IOBUF_X78_Y7_N32</td>
</tr>
<tr>
<td>-ALTEGRA DATA0~</td>
<td>-ALTEGRA DATA0~--CLUSTER</td>
<td>I/O pin cluster</td>
<td>AU27</td>
</tr>
<tr>
<td>-ALTEGRA DATA0~--obuf</td>
<td>-ALTEGRA DATA0~--CLUSTER</td>
<td>I/O pad</td>
<td>PIN_AU27</td>
</tr>
<tr>
<td>GCLK OR SMALLER REGION--CLKMUX GR</td>
<td>GCLK OR SMALLER REGION--CLKMUX GR</td>
<td>Clock core fanout</td>
<td>GLOBAL_CLOCK_REGION</td>
</tr>
<tr>
<td>GCLK OR SMALLER REGION--CLKMUX GR</td>
<td>Clock core fanout</td>
<td>GLOBAL_CLOCK_REGION</td>
<td></td>
</tr>
<tr>
<td>GCLK OR SMALLER REGION--CLKMUX GR</td>
<td>Clock core fanout</td>
<td>GLOBAL_CLOCK_REGION</td>
<td></td>
</tr>
<tr>
<td>GCLK OR SMALLER REGION--CLKMUX GR</td>
<td>Clock core fanout</td>
<td>GLOBAL_CLOCK_REGION</td>
<td></td>
</tr>
<tr>
<td>HSSI PLD INTERFACE</td>
<td>HSSI core interface</td>
<td>HSSI_PLD_INTERFACE</td>
<td></td>
</tr>
</tbody>
</table>

Report Cell Connectivity

Click the following Reports tab commands to generate reports showing the connections between all cells in the BluePrint plan.
Table 3-15: Report Cell Connectivity Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click &gt; Report Periphery Cell Connectivity</td>
<td>Right-click any Cell Name in the reports to Report Periphery Cell Connectivity. The report lists the source and destination ports and type of connections to the selected cell. Right-click any cell to report all connections to the cell.</td>
</tr>
</tbody>
</table>

Figure 3-18: Cell Connectivity Report

Document Revision History

This document has the following revision history.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integration of content into Quartus Prime Handbook.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added descriptions of new dynamic reports.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Package View description.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added GUI controls reference.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Second beta release of document on Molson. Added information about the following subjects:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Overview information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reset Plan command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Legal Assignments list and prompt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Tcl console</td>
</tr>
</tbody>
</table>
FPGA design software that easily integrates into your design flow saves time and improves productivity. The Altera Quartus Prime software provides you with a command-line executable for each step of the FPGA design flow to make the design process customizable and flexible.

The benefits provided by command-line executables include:

- Command-line control over each step of the design flow
- Reduced memory requirements
- Improved performance

The command-line executables are also completely interchangeable with the Quartus Prime GUI, allowing you to use the exact combination of tools that you prefer.

**Benefits of Command-Line Executables**

The Quartus Prime command-line executables provide control over each step of the design flow. Each executable includes options to control commonly used software settings. Each executable also provides detailed, built-in help describing its function, available options, and settings.

Command-line executables allow for easy integration with scripted design flows. You can easily create scripts with a series of commands. These scripts can be batch-processed, allowing for integration with distributed computing in server farms. These scripting capabilities enhance the ease of integration between the Quartus Prime software and other EDA synthesis, simulation, and verification software.

Command-line executables add flexibility without sacrificing the ease-of-use of the Quartus Prime GUI. You can use the Quartus Prime GUI and command-line executables at different stages in the design flow. For example, you might use the Quartus Prime GUI to edit the floorplan for the design, use the command-line executables to perform place-and-route, and return to the Quartus Prime GUI to perform debugging with the Chip Editor.

Command-line executables reduce the amount of memory required during each step in the design flow. Because each executable targets only one step in the design flow, the executables themselves are relatively compact, both in file size and the amount of memory used during processing. This memory usage reduction improves performance, and is particularly beneficial in design environments where heavy usage of computing resources results in reduced memory availability.

**Related Information**

*Using the Quartus Prime Executables in Shell Scripts*
INTRODUCTORY EXAMPLE

The following introduction to command-line executables demonstrates how to create a project, fit the
design, and generate programming files.

The tutorial design included with the Quartus Prime software is used to demonstrate this functionality. If
installed, the tutorial design is found in the <Quartus Prime directory>/qdesigns/fir_filter directory.

Before making changes, copy the tutorial directory and type the four commands shown in the introdutory
example below at a command prompt in the new project directory.

The <Quartus Prime directory>/quartus/bin directory must be in your PATH environment variable.

```
quartus_syn filtref --source=filtref.bdf --family=”Arria 10”
quartus_fit filtref --part=<part> --pack_register=minimize_area
quartus_asm filtref
quartus_sta filtref
```

The `quartus_syn filtref --source=filtref.bdf --family=”Arria 10”` command creates a new
Quartus Prime project called `filtref` with `filtref.bdf` as the top-level file. It targets the Arria 10 device family
and performs logic synthesis and technology mapping on the design files.

The `quartus_fit filtref --part=<part> --pack_register=minimize_area` command performs fitting on the `filtref`
project. This command specifies the device, and directs the Fitter to pack sequential and combinational
functions into single logic cells to reduce device resource usage.

The `quartus_asm filtref` command creates programming files for the `filtref` project.

The `quartus_sta filtref` command performs basic timing analysis on the `filtref` project using the
Quartus Prime TimeQuest Timing Analyzer, reporting worst-case setup slack, worst-case hold slack, and
other measurements.

You can put the four commands from the introductory example into a batch file or script file, and run
them. For example, you can create a simple UNIX shell script called `compile.sh`, which includes the code
shown in the UNIX shell script example below.

```
#!/bin/sh
PROJECT=filtref
TOP_LEVEL_FILE=filtref.bdf
FAMILY=”Arria 10”
PART=<part>
PACKING_OPTION=minimize_area
quartus_syn $PROJECT --source=$TOP_LEVEL_FILE --family=$FAMILY
quartus_fit $PROJECT --part=$PART --pack_register=$PACKING_OPTION
quartus_asm $PROJECT
quartus_sta $PROJECT
```

Edit the script as necessary and compile your project.

Related Information

**TimeQuest Timing Analyzer Quick Start Tutorial**
For more information about using all of the features of the quartus_sta executable. The TimeQuest
Timing Analyzer employs Synopsys Design Constraints to fully analyze the timing of your design.
Command-Line Scripting Help

Help for command-line executables is available through different methods. You can access help built in to the executables with command-line options. You can use the Quartus Prime Command-Line and Tcl API Help browser for an easy graphical view of the help information.

To use the Quartus Prime Command-Line and Tcl API Help browser, type the following command:

```
quartus_sh --qhelp
```

This command starts the Quartus Prime Command-Line and Tcl API Help browser, a viewer for information about the Quartus Prime Command-Line executables and Tcl API.

Use the `-h` option with any of the Quartus Prime Command-Line executables to get a description and list of supported options. Use the `--help=<option name>` option for detailed information about each option.

Figure 4-1: Quartus Prime Command-Line and Tcl API Help Browser

---

Project Settings with Command-Line Options

Command-line options are provided for many common global project settings and for performing common tasks.

You can use either of two methods to make assignments to an individual entity. If the project exists, open the project in the Quartus Prime GUI, change the assignment, and close the project. The changed assignment is updated in the `.qsf`. Any command-line executables that are run after this update use the
updated assignment. You can also make assignments using the Quartus Prime Tcl scripting API. If you want to completely script the creation of a Quartus Prime project, choose this method.

Related Information
- Option Precedence on page 4-4
- Tcl Scripting on page 5-1
- QSF Reference Manual

Option Precedence
If you use command-line executables, you must be aware of the precedence of various project assignments and how to control the precedence. Assignments for a particular project exist in the Quartus Prime Settings File (.qsf) for the project. Before the .qsf is updated after assignment changes, the updated assignments are reflected in compiler database files that hold intermediate compilation results.

All command-line options override any conflicting assignments found in the .qsf or the compiler database files. There are two command-line options to specify whether the .qsf or compiler database files take precedence for any assignments not specified as command-line options.

Any assignment not specified as a command-line option or found in the .qsf or compiler database file is set to its default value.

The file precedence command-line options are --read_settings_files and --write_settings_files.

By default, the --read_settings_files and --write_settings_files options are turned on. Turning on the --read_settings_files option causes a command-line executable to read assignments from the .qsf instead of from the compiler database files. Turning on the --write_settings_files option causes a command-line executable to update the .qsf to reflect any specified options, as happens when you close a project in the Quartus Prime GUI.

If you use command-line executables, be aware of the precedence of various project assignments and how to control the precedence. Assignments for a particular project can exist in three places:

- The .qsf for the project
- The result of the last compilation, in the /db directory, which reflects the assignments that existed when the project was compiled
- Command-line options

The precedence for reading assignments depends on the value of the --read_settings_files option.

Table 4-1: Precedence for Reading Assignments

<table>
<thead>
<tr>
<th>Option Specified</th>
<th>Precedence for Reading Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>--read_settings_files = on</td>
<td>Command-line options</td>
</tr>
<tr>
<td>(Default)</td>
<td>The .qsf for the project</td>
</tr>
<tr>
<td></td>
<td>Project database (db directory, if it exists)</td>
</tr>
<tr>
<td></td>
<td>Quartus Prime software defaults</td>
</tr>
<tr>
<td>Option Specified</td>
<td>Precedence for Reading Assignments</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--read_settings_files = off</td>
<td>Command-line options</td>
</tr>
<tr>
<td></td>
<td>Project database (db directory, if it exists)</td>
</tr>
<tr>
<td></td>
<td>Quartus Prime software defaults</td>
</tr>
</tbody>
</table>

The table lists the locations to which assignments are written, depending on the value of the --write_settings_files command-line option.

### Table 4-2: Location for Writing Assignments

<table>
<thead>
<tr>
<th>Option Specified</th>
<th>Location for Writing Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>--write_settings_files = on (Default)</td>
<td>.qsf and compiler database</td>
</tr>
<tr>
<td>--write_settings_files = off</td>
<td>Compiler database</td>
</tr>
</tbody>
</table>

The example assumes that a project named `fir_filter` exists, and that the analysis and synthesis step has been performed.

```
quartus_fit fir_filter --pack_register=off
quartus_sta fir_filter
mv fir_filter_sta.rpt fir_filter_1_sta.rpt
quartus_fit fir_filter --pack_register=minimize_area --write_settings_files=off
quartus_sta fir_filter
mv fir_filter_sta.rpt fir_filter_2_sta.rpt
```

The first command, `quartus_fit fir_filter --pack_register=off`, runs the `quartus_fit` executable with no aggressive attempts to reduce device resource usage.

The second command, `quartus_sta fir_filter`, performs basic timing analysis for the results of the previous fit.

The third command uses the UNIX `mv` command to copy the report file output from `quartus_sta` to a file with a new name, so that the results are not overwritten by subsequent timing analysis.

The fourth command runs `quartus_fit` a second time, and directs it to attempt to pack logic into registers to reduce device resource usage. With the --write_settings_files=off option, the command-line executable does not update the `.qsf` to reflect the changed register packing setting. Instead, only the compiler database files reflect the changed setting. If the --write_settings_files=off option is not specified, the command-line executable updates the `.qsf` to reflect the register packing setting.

The fifth command reruns timing analysis, and the sixth command renames the report file, so that it is not overwritten by subsequent timing analysis.

Use the options --read_settings_files=off and --write_settings_files=off (where appropriate) to optimize the way that the Quartus Prime software reads and updates settings files. In the following
example, the `quartus_asm` executable does not read or write settings files because doing so would not change any settings for the project.

```bash
quartus_syn filtref --source=filtref --part=<part>
quartus_fit filtref --pack_register=off --read_settings_files=off
quartus_asm filtref --read_settings_files=off --write_settings_files=off
```

## Compilation with `quartus_sh` --flow

The figure shows a typical Quartus Prime FPGA design flow using command-line executables.

**Figure 4-2: Typical Design Flow**

Use the `quartus_sh` executable with the `--flow` option to perform a complete compilation flow with a single command. The `--flow` option supports the smart recompile feature and efficiently sets command-line arguments for each executable in the flow.
The following example runs compilation, timing analysis, and programming file generation with a single command:

quartus_sh --flow compile filtref

**Tip:** For information about specialized flows, type `quartus_sh --help=flow` at a command prompt.

### Text-Based Report Files

Each command-line executable creates a text report file when it is run. These files report success or failure, and contain information about the processing performed by the executable.

Report file names contain the revision name and the short-form name of the executable that generated the report file, in the format `<revision>.<executable>.rpt`. For example, using the `quartus_fit` executable to place and route a project with the revision name `design_top` generates a report file named `design_top.fit.rpt`. Similarly, using the `quartus_sta` executable to perform timing analysis on a project with the revision name `fir_filter` generates a report file named `fir_filter.sta.rpt`.

As an alternative to parsing text-based report files, you can use the `::quartus::report` Tcl package.

**Related Information**

- [Tcl Scripting](#) on page 5-1

### Using Command-Line Executables In Scripts

You can use command-line executables in scripts that control other software in addition to the Quartus Prime software. For example, if your design flow uses third-party synthesis or simulation software, and if you can run the other software at a command prompt, you can include those commands with Quartus Prime executables in a single script.

The Quartus Prime command-line executables include options for common global project settings and operations, but you must use a Tcl script or the Quartus Prime GUI to set up a new project and apply individual constraints, such as pin location assignments and timing requirements. Command-line executables are very useful for working with existing projects, for making common global settings, and for performing common operations. For more flexibility in a flow, use a Tcl script, which makes it easier to pass data between different stages of the design flow and have more control during the flow.

For example, a UNIX shell script could run other synthesis software, then place-and-route the design in the Quartus Prime software, then generate output netlists for other simulation software. This script shows a script that synthesizes a design with the Synopsys Synplify software, simulates the design using the Mentor Graphics ModelSim® software, and then compiles the design targeting a Cyclone V device.

```bash
#!/bin/sh
# Run synthesis first.
# This example assumes you use Synplify software
synplify -batch synthesize.tcl
# If your Quartus Prime project exists already, you can just
# recompile the design.
# You can also use the script described in a later example to
# create a new project from scratch
quartus_sh --flow compile myproject
# Use the quartus_sta executable to do fast and slow-model
# timing analysis
```
Common Scripting Examples

You can create scripts including command line executable to control common Quartus Prime processes.

Create a Project and Apply Constraints

The command-line executables include options for common global project settings and commands. To apply constraints such as pin locations and timing assignments, run a Tcl script with the constraints in it. You can write a Tcl constraint file yourself, or generate one for an existing project.

From the Project menu, click **Generate Tcl File for Project**.

The example creates a project with a Tcl script and applies project constraints using the tutorial design files in the Quartus Prime <installation directory>/qdesigns/fir_filter/ directory.

```
project_new filtref -overwrite
# Assign family, device, and top-level file
set_global_assignment -name FAMILY Arria 10
set_global_assignment -name DEVICE <Device>
set_global_assignment -name BDF_FILE filtref.bdf
# Assign pins
set_location_assignment -to clk Pin_28
set_location_assignment -to clksx2 Pin_29
set_location_assignment -to d[0] Pin_139
set_location_assignment -to d[1] Pin_140
# Other assignments could follow
project_close
```

Save the script in a file called **setup_proj.tcl** and type the commands illustrated in the example at a command prompt to create the design, apply constraints, compile the design, and perform fast-corner and slow-corner timing analysis. Timing analysis results are saved in two files, filtref_sta_1.rpt and filtref_sta_2.rpt.

```
quartus_sh -t setup_proj.tcl
quartus_map filtref
quartus_fit filtref
quartus_asm filtref
quartus_sta filtref --model=fast --export_settings=off
mv filtref_sta.rpt filtref_sta_1.rpt
quartus_sta filtref --export_settings=off
mv filtref_sta.rpt filtref_sta_2.rpt
```
Type the following commands to create the design, apply constraints, and compile the design, without performing timing analysis:

```
quartus_sh -t setup_proj.tcl
quartus_sh --flow compile filtref
```

The `quartus_sh --flow compile` command performs a full compilation, and is equivalent to clicking the **Start Compilation** button in the toolbar.

### Check Design File Syntax

The UNIX shell script example shown in the example assumes that the Quartus Prime software **fir_filter** tutorial project exists in the current directory. You can find the **fir_filter** project in the `<Quartus Prime directory>/qdesigns/fir_filter` directory unless the Quartus Prime software tutorial files are not installed.

The `--analyze_file` option causes the `quartus_syn` executable to perform a syntax check on each file.

The script checks the exit code of the `quartus_map` executable to determine whether there is an error during the syntax check. Files with syntax errors are added to the `FILES_WITH_ERRORS` variable, and when all files are checked, the script prints a message indicating syntax errors.

When options are not specified, the executable uses the project database values. If not specified in the project database, the executable uses the Quartus Prime software default values. For example, the **fir_filter** project is set to target the Cyclone device family, so it is not necessary to specify the `--family` option.

```bash
#!/bin/sh
FILES_WITH_ERRORS=""
# Iterate over each file with a .bdf or .v extension
for filename in `ls *.bdf *.v`
do
  # Perform a syntax check on the specified file
  quartus_syn fir_filter --analyze_file=$filename
  # If the exit code is non-zero, the file has a syntax error
  if [ $? -ne 0 ]
    then
      FILES_WITH_ERRORS="$FILES_WITH_ERRORS $filename"
  fi
done
if [ -z "$FILES_WITH_ERRORS" ]
then
  echo "All files passed the syntax check"
  exit 0
else
  echo "There were syntax errors in the following file(s)"
  echo $FILES_WITH_ERRORS
  exit 1
fi
```

### Create a Project and Synthesize a Netlist Using Netlist Optimizations

This example creates a new Quartus Prime project with a file **top.edf** as the top-level entity. The `--enable_register_retiming=on` and `--enable_wysiwyg_resynthesis=on` options cause `quartus_map` to optimize the design using gate-level register retiming and technology remapping.

The `--part` option causes `quartus_syn` to target a device. To create the project and synthesize it using the netlist optimizations described above, type the command shown in this example at a command prompt.

```
quartus_syn top --source=top.edf --enable_register_retiming=on
  --enable_wysiwyg_resynthesis=on --part=<part>
```
Archive and Restore Projects

You can archive or restore a Quartus Prime Archive File (.qar) with a single command. This makes it easy to take snapshots of projects when you use batch files or shell scripts for compilation and project management.

Use the `--archive` or `--restore` options for `quartus_sh` as appropriate. Type the command shown in the example at a command prompt to archive your project.

```bash
quartus_sh --archive <project name>
```

The archive file is automatically named `<project name>.qar`. If you want to use a different name, type the command with the `-output` option as shown in example the example.

```bash
quartus_sh --archive <project name> -output <filename>
```

To restore a project archive, type the command shown in the example at a command prompt.

```bash
quartus_sh --restore <archive name>
```

The command restores the project archive to the current directory and overwrites existing files.

Related Information

Managing Quartus Prime Projects

Update Memory Contents Without Recompiling

You can use two commands to update the contents of memory blocks in your design without recompiling. Use the `quartus_cdb` executable with the `--update_mif` option to update memory contents from .mif or .hexout files. Then, rerun the assembler with the `quartus_asm` executable to regenerate the .sof, .pof, and any other programming files.

```bash
quartus_cdb --update_mif <project name> [--rev=<revision name>]
quartus_asm <project name> [--rev=<revision name>]
```

The example shows the commands for a DOS batch file for this example. With a DOS batch file, you can specify the project name and the revision name once for both commands. To create the DOS batch file, paste the following lines into a file called `update_memory.bat`.

```bash
quartus_cdb --update_mif %1 --rev=%2
quartus_asm %1 --rev=%2
```

To run the batch file, type the following command at a command prompt:

```bash
update_memory.bat <project name> <revision name>
```

Create a Compressed Configuration File

You can create a compressed configuration file in two ways. The first way is to run `quartus_cpf` with an option file that turns on compression.

To create an option file that turns on compression, type the following command at a command prompt:

```bash
quartus_cpf -w <filename>.opt```
This interactive command guides you through some questions, then creates an option file based on your answers. Use \--option to cause \texttt{quartus_cpf} to use the option file. For example, the following command creates a compressed \texttt{.pof} that targets an EPCS64 device:

\begin{verbatim}
quartus_cpf --convert --option=\texttt{filename}.opt --device=EPCS64 \texttt{file}.sof \texttt{file}.pof
\end{verbatim}

Alternatively, you can use the Convert Programming Files utility in the Quartus Prime software GUI to create a Conversion Setup File (\texttt{.cof}). Configure any options you want, including compression, then save the conversion setup. Use the following command to run the conversion setup you specified.

```
quartus_cpf --convert \texttt{file}.cof
```

\section*{Fit a Design as Quickly as Possible}

This example assumes that a project called \texttt{top} exists in the current directory, and that the name of the top-level entity is \texttt{top}. The \texttt{--effort=fast} option causes the \texttt{quartus_fit} to use the fast fit algorithm to increase compilation speed, possibly at the expense of reduced \(f_{\text{MAX}}\) performance. The \texttt{--one_fit_attempt=on} option restricts the Fitter to only one fitting attempt for the design.

To attempt to fit the project called \texttt{top} as quickly as possible, type the command shown at a command prompt.

```
quartus_fit top --effort=fast --one_fit_attempt=on
```

\section*{Fit a Design Using Multiple Seeds}

This shell script example assumes that the Quartus Prime software tutorial project called \texttt{fir_filter} exists in the current directory (defined in the file \texttt{fir_filter.qpf}). If the tutorial files are installed on your system, this project exists in the \texttt{<Quartus Prime directory>/qdesigns<quartus_version_number>/fir_filter} directory.

Because the top-level entity in the project does not have the same name as the project, you must specify the revision name for the top-level entity with the \texttt{--rev} option. The \texttt{--seed} option specifies the seeds to use for fitting.

A seed is a parameter that affects the random initial placement of the Quartus Prime Fitter. Varying the seed can result in better performance for some designs.

After each fitting attempt, the script creates new directories for the results of each fitting attempt and copies the complete project to the new directory so that the results are available for viewing and debugging after the script has completed.

```
#!/bin/sh
ERROR_SEEDS=""
quartus_map fir_filter --rev=filtref
# Iterate over a number of seeds
for seed in 1 2 3 4 5
  do
    echo "Starting fit with seed=$seed"
    # Perform a fitting attempt with the specified seed
    quartus_fit fir_filter --seed=$seed --rev=filtref
    # If the exit-code is non-zero, the fitting attempt was
    # successful, so copy the project to a new directory
    if [ $? -eq 0 ]
      then
        mkdir ../fir_filter-seed_$seed
        mkdir ../fir_filter-seed_$seed/db
        cp * ../fir_filter-seed_$seed
```

\section*{Command Line Scripting}

\begin{tabular}{l}
Alterna Corporation
\end{tabular}

\begin{tabular}{l}
Send Feedback
\end{tabular}
cp db/* ../fir_filter-seed_$seed/db
else
ERROR_SEEDS="$ERROR_SEEDS $seed"
fi
done
if [ -z "$ERROR_SEEDS" ]
then
echo "Seed sweeping was successful"
exit 0
else
echo "There were errors with the following seed(s)"
echo $ERROR_SEEDS
exit 1
fi

#!/bin/sh
ERROR_SEEDS=""
quartus_syn fir_filter --rev=filtref
# Iterate over a number of seeds
for seed in 1 2 3 4 5
do
echo "Starting fit with seed=$seed"
# Perform a fitting attempt with the specified seed
quartus_fit fir_filter --seed=$seed --rev=filtref
# If the exit-code is non-zero, the fitting attempt was
# successful, so copy the project to a new directory
if [ $? -eq 0 ]
then
mkdir ../fir_filter-seed_$seed
mkdir ../fir_filter-seed_$seed/db
cp * ../fir_filter-seed_$seed
else
ERROR_SEEDS="$ERROR_SEEDS $seed"
fi
done
if [ -z "$ERROR_SEEDS" ]
then
echo "Seed sweeping was successful"
exit 0
else
echo "There were errors with the following seed(s)"
echo $ERROR_SEEDS
exit 1
fi

Tip: Use Design Space Explorer II (DSE) included with the Quartus Prime software script (by typing
quartus_dse at a command prompt) to improve design performance by performing automated seed
sweeping.

The QFlow Script

A Tcl/Tk-based graphical interface called QFlow is included with the command-line executables. You can
use the QFlow interface to open projects, launch some of the command-line executables, view report files,
and make some global project assignments.
The QFlow interface can run the following command-line executables:

- quartus_syn (Analysis and Synthesis)
- quartus_fit (Fitter)
- quartus_sta (TimeQuest timing analyzer)
- quartus_asm (Assembler)
- quartus_eda (EDA Netlist Writer)

To view floorplans or perform other GUI-intensive tasks, launch the Quartus Prime software.

Start QFlow by typing the following command at a command prompt:

```
quartus_sh -g
```

**Tip:** The QFlow script is located in the `<Quartus Prime directory>/common/tcl/apps/qflow/` directory.

---

### Document Revision History

**Table 4-3: Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Remove descriptions of makefile support that was removed from software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in 14.0.</td>
</tr>
<tr>
<td>December</td>
<td>14.1.0</td>
<td>Updated DSE II commands.</td>
</tr>
<tr>
<td>2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated formatting.</td>
</tr>
<tr>
<td>November</td>
<td>13.1.0</td>
<td>Removed information about -silnet qmegawiz command</td>
</tr>
<tr>
<td>2013</td>
<td></td>
<td></td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November</td>
<td>11.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Corrected quartus_qpf example usage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated examples.</td>
</tr>
<tr>
<td>December</td>
<td>10.1.0</td>
<td>Template update.</td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td>Added section on using a script to regenerate megafunction variations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed references to the Classic Timing Analyzer (quartus_tan).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed Qflow illustration.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Updated script examples to use quartus_sta instead of quartus_tan, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other minor updates throughout document.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Updated Table 2–1 to add quartus_jli and quartus_jbcc executables and descriptions, and other minor updates throughout document.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Added the following sections:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “The MegaWizard Plug-In Manager” on page 2–11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Command-Line Support” on page 2–12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Module and Wizard Names” on page 2–13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Ports and Parameters” on page 2–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Invalid Configurations” on page 2–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Strategies to Determine Port and Parameter Values” on page 2–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Optional Files” on page 2–15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Parameter File” on page 2–16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Working Directory” on page 2–17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Variation File Name” on page 2–17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “Create a Compressed Configuration File” on page 2–21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Option Precedence” on page 2–5 to clarify how to control precedence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected Example 2–5 on page 2–8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed Example 2–1, Example 2–2, Example 2–4, and Example 2–7 to use the EP1C12F256C6 device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated entire chapter using 8½” × 11” chapter template</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Updated “Referenced Documents” on page 2–20.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated references in document.</td>
</tr>
</tbody>
</table>

**Related Information**

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

You can use Tcl scripts to control the Altera® Quartus Prime software and to perform a wide range of functions, such as compiling a design or scripting common tasks.

For example, use Tcl scripts to perform the following tasks:

- Manage a Quartus Prime project
- Make assignments
- Define design constraints
- Make device assignments
- Compile your design
- Perform timing analysis
- Access reports

Tcl scripts also facilitate project or assignment migration. For example, when designing in different projects with the same prototype or development board, you can write a script to automate reassignment of pin locations in each new project. The Quartus Prime software can also generate a Tcl script based on all the current assignments in the project, which aids in switching assignments to another project.

The Quartus Prime software Tcl commands follow the EDA industry Tcl application programming interface (API) standards for command-line options. This simplifies learning and using Tcl commands. If you encounter an error with a command argument, the Tcl interpreter includes help information showing correct usage.

This chapter includes sample Tcl scripts for automating tasks in the Quartus Prime software. You can modify these example scripts for use with your own designs. You can find more Tcl scripts in the Design Examples section of the Support area on the Altera website.

Related Information

Design Examples page on the Altera website
Tool Command Language

Tcl (pronounced “tickle”) stands for Tool Command Language, a popular scripting language that is similar to many shell scripting and high-level programming languages. It provides support for control structures, variables, network socket access, and APIs.

Synopsys, Mentor Graphics®, and Altera software products support the Tcl industry-standard scripting language. Tcl allows you to create custom commands and works seamlessly across most development platforms.

You can create your own procedures by writing scripts combining basic Tcl commands and Quartus Prime API functions. You can then automate your design flow, run the Quartus Prime software in batch mode, or execute the individual Tcl commands interactively in the Quartus Prime Tcl interactive shell.

The Quartus Prime software supports Tcl/Tk version 8.5, supplied by the Tcl DeveloperXchange.

Related Information

- **External References** on page 5-22
  For a list of recommended literature on Tcl.
- **Tcl Scripting Basics** on page 5-17
  For more information on Tcl scripting, or if you are a Tcl beginner.
- [tcl.activestate.com/](http://tcl.activestate.com/)

### Quartus Prime Tcl Packages

The Quartus Prime software groups Tcl commands into packages by function.

**Table 5-1: Quartus Prime Tcl Packages**

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Package Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chip_planner</td>
<td>Identify and modify resource usage and routing with the Chip Editor</td>
</tr>
<tr>
<td>design</td>
<td>Manipulate project databases, including the assignments database, to enable the creation of instance assignments without modifying the .qsf file</td>
</tr>
<tr>
<td>device</td>
<td>Get device and family information from the device database</td>
</tr>
<tr>
<td>external_memif_toolkit</td>
<td>Interact with external memory interfaces and debug components</td>
</tr>
<tr>
<td>fif</td>
<td>Contains the set of Tcl functions for using the Fault Injection File (FIF) Driver</td>
</tr>
<tr>
<td>flow</td>
<td>Compile a project, run command-line executables, and other common flows</td>
</tr>
<tr>
<td>insystem_memory_edit</td>
<td>Read and edit memory contents in Altera devices</td>
</tr>
<tr>
<td>insystem_source_probe</td>
<td>Interact with the In-System Sources and Probes tool in an Altera device</td>
</tr>
<tr>
<td>interactive_synthesis</td>
<td></td>
</tr>
<tr>
<td>Package Name</td>
<td>Package Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iptclgen</td>
<td>Generate memory IP</td>
</tr>
<tr>
<td>jtag</td>
<td>Control the JTAG chain</td>
</tr>
<tr>
<td>logic_analyzer_</td>
<td>Query and modify the Logic Analyzer Interface output pin state</td>
</tr>
<tr>
<td>interface</td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td>Perform miscellaneous tasks such as enabling natural bus naming, package loading, and message posting</td>
</tr>
<tr>
<td>names</td>
<td></td>
</tr>
<tr>
<td>periph</td>
<td>Interact with the BluePrint plans</td>
</tr>
<tr>
<td>project</td>
<td>Create and manage projects and revisions, make any project assignments including timing assignments</td>
</tr>
<tr>
<td>report</td>
<td>Get information from report tables, create custom reports</td>
</tr>
<tr>
<td>rtl</td>
<td>Traverse and query the RTL netlist of your design</td>
</tr>
<tr>
<td>rtm</td>
<td></td>
</tr>
<tr>
<td>sdc</td>
<td>Specify constraints and exceptions to the TimeQuest Timing Analyzer</td>
</tr>
<tr>
<td>sdc_ext</td>
<td>Altera-specific SDC commands</td>
</tr>
<tr>
<td>simulator</td>
<td>Configure and perform simulations</td>
</tr>
<tr>
<td>sta</td>
<td>Contain the set of Tcl functions for obtaining advanced information from the TimeQuest Timing Analyzer</td>
</tr>
<tr>
<td>stp</td>
<td>Run the SignalTap® II Logic Analyzer</td>
</tr>
<tr>
<td>synthesis_report</td>
<td>Contain the set of Tcl functions for the Dynamic Synthesis Report tool</td>
</tr>
<tr>
<td>tdc</td>
<td>Obtain information from the TimeQuest Timing Analyzer</td>
</tr>
</tbody>
</table>

By default, only the minimum number of packages loads automatically with each Quartus Prime executable. This keeps the memory requirement for each executable as low as possible. Because the number of packages that the Quartus Prime executable loads is limited, you must load other packages before you can run commands in those packages.

Because different packages are available in different executables, you must run your scripts with executables that include the packages you use in the scripts. For example, if you use commands in the sdc_ext package, you must use the quartus_sta executable to run the script because the quartus_sta executable is the only one with support for the sdc_ext package.

The following command prints lists of the packages loaded or available to load for an executable, to the console:

```
<executable name> --tcl_eval help
```

For example, type the following command to list the packages loaded or available to load by the quartus_fit executable:

```
quartus_fit --tcl_eval help
```
Loading Packages

To load a Quartus Prime Tcl package, use the `load_package` command as follows:

```tcl
load_package [-version <version number>] <package name>
```

This command is similar to the `package require` Tcl command, but you can easily alternate between different versions of a Quartus Prime Tcl package with the `load_package` command because of the `-version` option.

Related Information

- **Command-Line Scripting** on page 4-1
  For additional information about these and other Quartus Prime command-line executables.

Quartus Prime Tcl API Help

Access the Quartus Prime Tcl API Help reference by typing the following command at a system command prompt:

```bash
quartus_sh --qhelp
```

This command runs the Quartus Prime Command-Line and Tcl API help browser, which documents all commands and options in the Quartus Prime Tcl API.

Quartus Prime Tcl help allows easy access to information about the Quartus Prime Tcl commands. To access the help information, type `help` at a Tcl prompt.

Tcl Help Output

### Table 5-2: Help Options Available in the Quartus Prime Tcl Environment

<table>
<thead>
<tr>
<th>Help Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>help</code></td>
<td>To view a list of available Quartus Prime Tcl packages, loaded and not loaded.</td>
</tr>
<tr>
<td><code>help -tcl</code></td>
<td>To view a list of commands used to load Tcl packages and access command-line help.</td>
</tr>
<tr>
<td><code>help -pkg &lt;package_name&gt;</code> [-version &lt;version number&gt;]</td>
<td>To view help for a specified Quartus Prime package that includes the list of available Tcl commands. For convenience, you can omit the <code>::quartus::</code> package prefix, and type <code>help -pkg &lt;package_name&gt;</code>. If you do not specify the <code>-version</code> option, help for the currently loaded package is displayed by default. If the package for which you want help is not loaded, help for the latest version of the package is displayed by default. Examples: <code>help -pkg ::quartus::project</code> <code>help -pkg project help -pkg project -version 1.0</code></td>
</tr>
<tr>
<td>Help Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>&lt;command_name&gt; -h or &lt;command_name&gt; -help</td>
<td>To view short help for a Quartus Prime Tcl command for which the package is loaded. Examples: project_open -h project_open -help</td>
</tr>
<tr>
<td>package require ::quartus::&lt;package name&gt; [&lt;version&gt;]</td>
<td>To load a Quartus Prime Tcl package with the specified version. If &lt;version&gt; is not specified, the latest version of the package is loaded by default. Example: package require ::quartus::project 1.0 This command is similar to the load_package command. The advantage of the load_package command is that you can alternate freely between different versions of the same package. Type load_package &lt;package name&gt; [-version &lt;version number&gt;] to load a Quartus Prime Tcl package with the specified version. If the -version option is not specified, the latest version of the package is loaded by default. Example: load_package ::quartus::project -version 1.0</td>
</tr>
<tr>
<td>help -cmd &lt;command_name&gt; [-version &lt;version&gt;] or &lt;command_name&gt; -long_help</td>
<td>To view complete help text for a Quartus Prime Tcl command. If you do not specify the -version option, help for the command in the currently loaded package version is displayed by default. If the package version for which you want help is not loaded, help for the latest version of the package is displayed by default. Examples: project_open -long_help help -cmd project_open help -cmd project_open -version 1.0</td>
</tr>
<tr>
<td>help -examples</td>
<td>To view examples of Quartus Prime Tcl usage.</td>
</tr>
<tr>
<td>help -quartus</td>
<td>To view help on the predefined global Tcl array that contains project information and information about the Quartus Prime executable that is currently running.</td>
</tr>
<tr>
<td>quartus_sh --qhelp</td>
<td>To launch the Tk viewer for Quartus Prime command-line help and display help for the command-line executables and Tcl API packages.</td>
</tr>
<tr>
<td>help -timequestinfo</td>
<td>To view help on the predefined global &quot;TimeQuestInfo&quot; Tcl array that contains delay model information and speed grade information of a TimeQuest design that is currently running.</td>
</tr>
</tbody>
</table>
The Tcl API help is also available in Quartus Prime online help. Search for the command or package name to find details about that command or package.

**Related Information**

- [Command-Line Scripting](#) on page 4-1
  For more information about the Tk viewer for Quartus Prime command-line help.

## Command-Line Options: -t, -s, and --tcl_eval

There are three command-line options you can use with executables that support Tcl.

### Table 5-3: Command-Line Options Supporting Tcl Scripting

<table>
<thead>
<tr>
<th>Command-Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--script=/&lt;script file&gt; [/&lt;script args&gt;]</td>
<td>Run the specified Tcl script with optional arguments.</td>
</tr>
<tr>
<td>-t &lt;script file&gt; [/&lt;script args&gt;]</td>
<td>Run the specified Tcl script with optional arguments. The -t option is the short form of the --script option.</td>
</tr>
<tr>
<td>--shell</td>
<td>Open the executable in the interactive Tcl shell mode.</td>
</tr>
<tr>
<td>-s</td>
<td>Open the executable in the interactive Tcl shell mode. The -s option is the short form of the --shell option.</td>
</tr>
<tr>
<td>--tcl_eval &lt;tcl command&gt;</td>
<td>Evaluate the remaining command-line arguments as Tcl commands. For example, the following command displays help for the project package: quartus_sh --tcl_eval help -pkg project</td>
</tr>
</tbody>
</table>

### Run a Tcl Script

Running an executable with the -t option runs the specified Tcl script. You can also specify arguments to the script. Access the arguments through the argv variable, or use a package such as `cmdline`, which supports arguments of the following form:

- `<argument name> <argument value>`

The `cmdline` package is included in the `<Quartus Prime directory>/common/tcl/packages` directory.

For example, to run a script called `myscript.tcl` with one argument, `Stratix`, type the following command at a system command prompt:

```
quartus_sh -t myscript.tcl Stratix
```

### Interactive Shell Mode

Running an executable with the -s option starts an interactive Tcl shell. For example, to open the Quartus Prime TimeQuest Timing Analyzer executable in interactive shell mode, type the following command:

```
quartus_sta -s
```
Commands you type in the Tcl shell are interpreted when you press Enter. You can run a Tcl script in the interactive shell with the following command:

```
source <script name>
```

If a command is not recognized by the shell, it is assumed to be an external command and executed with the `exec` command.

**Evaluate as Tcl**

Running an executable with the `--tcl_eval` option causes the executable to immediately evaluate the remaining command-line arguments as Tcl commands. This can be useful if you want to run simple Tcl commands from other scripting languages.

For example, the following command runs the Tcl command that prints out the commands available in the project package.

```
quartus_sh --tcl_eval help -pkg project
```

**The Quartus Prime Tcl Console Window**

You can run Tcl commands directly in the Quartus Prime Tcl Console window. On the View menu, click **Utility Windows**. By default, the Tcl Console window is docked in the bottom-right corner of the Quartus Prime GUI. All Tcl commands typed in the Tcl Console are interpreted by the Quartus Prime Tcl shell.

**Note:** Some shell commands such as `cd`, `ls`, and others can be run in the Tcl Console window, with the Tcl `exec` command. However, for best results, run shell commands and Quartus Prime executables from a system command prompt outside of the Quartus Prime software GUI.

Tcl messages appear in the **System** tab (Messages window). Errors and messages written to `stdout` and `stderr` also are shown in the Quartus Prime Tcl Console window.

**End-to-End Design Flows**

You can use Tcl scripts to control all aspects of the design flow, including controlling other software, when the other software also includes a scripting interface.

Typically, EDA tools include their own script interpreters that extend core language functionality with tool-specific commands. For example, the Quartus Prime Tcl interpreter supports all core Tcl commands, and adds numerous commands specific to the Quartus Prime software. You can include commands in one Tcl script to run another script, which allows you to combine or chain together scripts to control different tools. Because scripts for different tools must be executed with different Tcl interpreters, it is difficult to pass information between the scripts unless one script writes information into a file and another script reads it.

Within the Quartus Prime software, you can perform many different operations in a design flow (such as synthesis, fitting, and timing analysis) from a single script, making it easy to maintain global state information and pass data between the operations. However, there are some limitations on the operations you can perform in a single script due to the various packages supported by each executable.

There are no limitations on running flows from any executable. Flows include operations found in the Start section of the Processing menu in the Quartus Prime GUI, and are also documented as options for
the `execute_flow` Tcl command. If you can make settings in the Quartus Prime software and run a flow to get your desired result, you can make the same settings and run the same flow in a Tcl script.

Creating Projects and Making Assignments

You can create a script that makes all the assignments for an existing project, and then use the script at any time to restore your project settings to a known state. From the Project menu, click **Generate Tcl File for Project** to automatically generate a `.tcl` file containing your assignments. You can source this file to recreate your project, and you can add other commands to this file, such as commands for compiling the design. The file is a good starting point to learn about project management commands and assignment commands.

The following example creates a project, makes assignments, and compiles the project. The example uses the `fir_filter` tutorial design files in the `qdesigns` installation directory. Run this script in the `fir_filter` directory, with the `quartus_sh` executable.

Create and Compile a Project

```tcl
load_package flow
# Create the project and overwrite any settings
# files that exist
project_new fir_filter -revision filtref -overwrite
# Set the device, the name of the top-level BDF, 
# and the name of the top level entity
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
set_global_assignment -name BDF_FILE filtref.bdf
set_global_assignment -name TOP_LEVEL_ENTITY filtref
# Add other pin assignments here
set_location_assignment -to clk Pin_G1
# compile the project
execute_flow -compile
project_close
```

**Note:** The assignments created or modified while a project is open are not committed to the Quartus Prime Settings File (`.qsf`) unless you explicitly call `export_assignments` or `project_close` (unless `-dont_export_assignments` is specified). In some cases, such as when running `execute_flow`, the Quartus Prime software automatically commits the changes.

Related Information

- [Interactive Shell Mode](#) on page 5-6
- [Constraining Designs](#) on page 1-1
  For more information on making assignments.
- [QSF Reference Manual](#)  
  For more information on scripting for all Quartus Prime project settings and assignments.

Compiling Designs

You can run the Quartus Prime command-line executables from Tcl scripts. Use the included `flow` package to run various Quartus Prime compilation flows, or run each executable directly.
The flow Package

The flow package includes two commands for running Quartus Prime command-line executables, either individually or together in standard compilation sequence. The execute_module command allows you to run an individual Quartus Prime command-line executable. The execute_flow command allows you to run some or all of the executables in commonly-used combinations. Use the flow package instead of system calls to run Quartus Prime executables from scripts or from the Quartus Prime Tcl Console.

Compile All Revisions

You can use a simple Tcl script to compile all revisions in your project. Save the following script in a file called compile_revisions.tcl and type the following to run it:

```
quartus_sh -t compile_revisions.tcl <project name>
```

```
load_package flow
project_open [lindex $quartus(args) 0]
set original_revision [get_current_revision]
foreach revision [get_project_revisions] {
    set_current_revision $revision
    execute flow -compile
}
set_current_revision $original_revision
project_close
```

Reporting

You can extract information from the Compilation Report to evaluate results. The Quartus Prime Tcl API provides easy access to report data so you do not have to write scripts to parse the text report files.

If you know the exact report cell or cells you want to access, use the get_report_panel_data command and specify the row and column names (or x and y coordinates) and the name of the appropriate report panel. You can often search for data in a report panel. To do this, use a loop that reads the report one row at a time with the get_report_panel_row command.

Column headings in report panels are in row 0. If you use a loop that reads the report one row at a time, you can start with row 1 to skip row 0 with column headings. The get_number_of_rows command returns the number of rows in the report panel, including the column heading row. Because the number of rows includes the column heading row, continue your loop as long as the loop index is less than the number of rows.

Report panels are hierarchically arranged and each level of hierarchy is denoted by the string "||" in the panel name. For example, the name of the Fitter Settings report panel is Fitter||Fitter Settings because it is in the Fitter folder. Panels at the highest hierarchy level do not use the "||" string. For example, the Flow Settings report panel is named Flow Settings.

The following Tcl code prints a list of all report panel names in your project. You can run this code with any executable that includes support for the report package.

```
load_package report
project_open myproject
```

Print All Report Panel Names
Viewing Report Data in Excel

The Microsoft Excel software can be useful in viewing and manipulating timing analysis results. You can create a Comma Separated Value (.csv) file from any Quartus Prime report to open with Excel. The following Tcl code shows a simple way to create a .csv file with data from the Fitter panel in a report. You could modify the script to use command-line arguments to pass in the name of the project, report panel, and output file to use. You can run this script example with any executable that supports the report package.

Create .csv Files from Reports

load_package report
project_open my-project
load_report
# This is the name of the report panel to save as a CSV file
set panel_name "Fitter||Fitter Settings"
set csv_file "output.csv"
set fh [open $csv_file w]
set num_rows [get_number_of_rows -name $panel_name]
# Go through all the rows in the report file, including the
# row with headings, and write out the comma-separated data
for { set i 0 } { $i < $num_rows } { incr i } {
    set row_data [get_report_panel_row -name $panel_name 
    -row $i]
    puts $fh [join $row_data ","]
}
close $fh
unload_report

Timing Analysis

The Quartus Prime TimeQuest Timing Analyzer includes support for industry-standard SDC commands in the sdc package. The Quartus Prime software includes comprehensive Tcl APIs and SDC extensions for the TimeQuest Timing Analyzer in the sta, and sdc_ext packages. The Quartus Prime software also includes a tdc package that obtains information from the TimeQuest Timing Analyzer.

Related Information
Quartus Prime TimeQuest Timing Analyzer
For information about how to perform timing analysis with the Quartus Prime TimeQuest Timing Analyzer

Automating Script Execution

You can configure scripts to run automatically at various points during compilation. Use this capability to automatically run scripts that perform custom reporting, make specific assignments, and perform many other tasks.
The following three global assignments control when a script is run automatically:

- **PRE_FLOW_SCRIPT_FILE** — before a flow starts
- **POST_MODULE_SCRIPT_FILE** — after a module finishes
- **POST_FLOW_SCRIPT_FILE** — after a flow finishes

A module is another term for a Quartus Prime executable that performs one step in a flow. For example, two modules are Analysis and Synthesis (`quartus_syn`), and timing analysis (`quartus_sta`).

A flow is a series of modules that the Quartus Prime software runs with predefined options. For example, compiling a design is a flow that typically consists of the following steps (performed by the indicated module):

1. Analysis and synthesis (`quartus_syn`)
2. Fitter (`quartus_fit`)
3. Assembler (`quartus_asm`)
4. Timing Analyzer (`quartus_sta`)

Other flows are described in the help for the `execute_flow` Tcl command. In addition, many commands in the Processing menu of the Quartus Prime GUI correspond to this design flow.

To make an assignment automatically run a script, add an assignment with the following form to the `.qsf` for your project:

```
set_global_assignment -name <assignment name> <executable>:<script name>
```

The Quartus Prime software runs the scripts.

```
<executable> -t <script name> <flow or module name> <project name> <revision name>
```

The first argument passed in the `argv` variable (or `quartus(args)` variable) is the name of the flow or module being executed, depending on the assignment you use. The second argument is the name of the project and the third argument is the name of the revision.

When you use the `POST_MODULE_SCRIPT_FILE` assignment, the specified script is automatically run after every executable in a flow. You can use a string comparison with the module name (the first argument passed in to the script) to isolate script processing to certain modules.

**Execution Example**

To illustrate how automatic script execution works in a complete flow, assume you have a project called `top` with a current revision called `rev_1`, and you have the following assignments in the `.qsf` for your project.

```
set_global_assignment -name PRE_FLOW_SCRIPT_FILE quartus_sh:first.tcl
set_global_assignment -name POST_MODULE_SCRIPT_FILE quartus_sh:next.tcl
set_global_assignment -name POST_FLOW_SCRIPT_FILE quartus_sh:last.tcl
```

When you compile your project, the `PRE_FLOW_SCRIPT_FILE` assignment causes the following command to be run before compilation begins:

```
quartus_sh -t first.tcl compile top rev_1
```

Next, the Quartus Prime software starts compilation with analysis and synthesis, performed by the `quartus_syn` executable. After the Analysis and Synthesis finishes, the `POST_MODULE_SCRIPT_FILE` assignment causes the following command to run:

```
quartus_sh -t next.tcl quartus_syn top rev_1
```
Then, the Quartus Prime software continues compilation with the Fitter, performed by the `quartus_fit` executable. After the Fitter finishes, the `POST_MODULE_SCRIPT_FILE` assignment runs the following command:

```
quartus_sh -t next.tcl quartus_fit top rev_1
```

Corresponding commands are run after the other stages of the compilation. When the compilation is over, the `POST_FLOW_SCRIPT_FILE` assignment runs the following command:

```
quartus_sh -t last.tcl compile top rev_1
```

### Controlling Processing

The `POST_MODULE_SCRIPT_FILE` assignment causes a script to run after every module. Because the same script is run after every module, you might have to include some conditional statements that restrict processing in your script to certain modules.

For example, if you want a script to run only after timing analysis, use a conditional test like the following example. It checks the flow or module name passed as the first argument to the script and executes code when the module is `quartus_sta`.

#### Restrict Processing to a Single Module

```tcl
set module [lindex $quartus(args) 0]
if [string match "quartus_sta" $module] {
    # Include commands here that are run after timing analysis
    # Use the post-message command to display messages
    post_message "Running after timing analysis"
}
```

### Displaying Messages

Because of the way the Quartus Prime software runs the scripts automatically, you must use the `post_message` command to display messages, instead of the `puts` command. This requirement applies only to scripts that are run by the three assignments listed in “Automating Script Execution”.

### Related Information

- **The post_message Command** on page 5-14
  For more information about this command.
- **Automating Script Execution** on page 5-10
  For more information on the three scripts capable of scripting-message automation.

### Other Scripting Features

The Quartus Prime Tcl API includes other general-purpose commands and features described in this section.

### Natural Bus Naming

The Quartus Prime software supports natural bus naming. Natural bus naming allows you to use square brackets to specify bus indexes in HDL without including escape characters to prevent Tcl from interpreting the square brackets as containing commands. For example, one signal in a bus named
address can be identified as address[0] instead of address\[0\]. You can take advantage of natural bus naming when making assignments.

```
set_location_assignment -to address[10] Pin_M20
```

The Quartus Prime software defaults to natural bus naming. You can turn off natural bus naming with the `disable_natural_bus_naming` command. For more information about natural bus naming, type the following at a Quartus Prime Tcl prompt:

```
enable_natural_bus_naming -h
```

### Short Option Names

You can use short versions of command options, as long as they are unambiguous. For example, the `project_open` command supports two options: `-current_revision` and `-revision`.

You can use any of the following abbreviations of the `-revision` option:

- `-r`
- `-re`
- `-rev`
- `-revi`
- `-revis`
- `-revisio`

You can use an option as short as `-r` because in the case of the `project_open` command no other option starts with the letter r. However, the `report_timing` command includes the options `-recovery` and `-removal`. You cannot use `-r` or `-re` to shorten either of those options, because the abbreviation would not be unique to only one option.

### Collection Commands

Some Quartus Prime Tcl functions return very large sets of data that would be inefficient as Tcl lists. These data structures are referred to as collections. The Quartus Prime Tcl API uses a collection ID to access the collection.

There are two Quartus Prime Tcl commands for working with collections, `foreach_in_collection` and `get_collection_size`. Use the `set` command to assign a collection ID to a variable.

**Related Information**

- `foreach_in_collection`  
  For information about which Quartus Prime Tcl commands return collection IDs.

**The `foreach_in_collection` Command**

The `foreach_in_collection` command is similar to the `foreach` Tcl command. Use it to iterate through all elements in a collection. The following example prints all instance assignments in an open project.

**foreach_in_collection Example**

```
set all_instance_assignments [get_all_instance_assignments -name *]
foreach_in_collection asgn $all_instance_assignments {
    # Information about each assignment is
    # returned in a list. For information
    # about the list elements, refer to Help
    # for the get-all-instance-assignments command.
}
The get_collection_size Command

Use the get_collection_size command to get the number of elements in a collection. The following example prints the number of global assignments in an open project.

get_collection_size Example

```tcl
set all_global_assignments [get_all_global_assignments -name *]
set num_global_assignments [get_collection_size $all_global_assignments]
puts "There are $num_global_assignments global assignments in your project"
```

The post_message Command

To print messages that are formatted like Quartus Prime software messages, use the post_message command. Messages printed by the post_message command appear in the System tab of the Messages window in the Quartus Prime GUI, and are written to standard at when scripts are run. Arguments for the post_message command include an optional message type and a required message string.

The message type can be one of the following:

- `info` (default)
- `extra_info`
- `warning`
- `critical_warning`
- `error`

If you do not specify a type, Quartus Prime software defaults to `info`.

With the Quartus Prime software in Windows, you can color code messages displayed at the system command prompt with the post_message command. Add the following line to your quartus2.ini file:

```ini
DISPLAY_COMMAND_LINE_MESSAGES_IN_COLOR = on
```

The following example shows how to use the post_message command.

```
post_message -type warning "Design has gated clocks"
```

Accessing Command-Line Arguments

Many Tcl scripts are designed to accept command-line arguments, such as the name of a project or revision. The global variable `quartus(args)` is a list of the arguments typed on the command-line following the name of the Tcl script. The following Tcl example prints all of the arguments in the `quartus(args)` variable.

Simple Command-Line Argument Access

```tcl
set i 0
foreach arg $quartus(args) {
    puts "The value at index $i is $arg"
}
```
If you copy the script in the previous example to a file named `print_args.tcl`, it displays the following output when you type the following at a command prompt.

**Passing Command-Line Arguments to Scripts**

```
quartus_sh -t print_args.tcl my_project 100MHz
The value at index 0 is my_project
The value at index 1 is 100MHz
```

**The cmdline Package**

You can use the `cmdline` package included with the Quartus Prime software for more robust and self-documenting command-line argument passing. The `cmdline` package supports command-line arguments with the form `-<option><value>.

### cmdline Package

```tcl
package require cmdline
variable ::argv0 $::quartus(args)
set options {
    { "project.arg" "Project name" }
    { "frequency.arg" "Frequency" }
}
set usage "You need to specify options and values"
array set optshash [::cmdline::getoptions ::argv0 $options $usage]
puts "The project name is $optshash(project)"
puts "The frequency is $optshash(frequency)"
```

If you save those commands in a Tcl script called `print_cmd_args.tcl` you see the following output when you type the following command at a command prompt.

**Passing Command-Line Arguments for Scripts**

```
quartus_sh -t print_cmd_args.tcl -project my_project -frequency 100MHz
The project name is my_project
The frequency is 100MHz
```

Virtually all Quartus Prime Tcl scripts must open a project. You can open a project, and you can optionally specify a revision name with code like the following example. The example checks whether the specified project exists. If it does, the example opens the current revision, or the revision you specify.

**Full-Featured Method to Open Projects**

```tcl
package require cmdline
variable ::argv0 $::quartus(args)
set options {
    { "project.arg" "Project Name" }
    { "revision.arg" "Revision Name" }
}
array set optshash [::cmdline::getoptions ::argv0 $options]
# Ensure the project exists before trying to open it
if {[project_exists $optshash(project)]} {
    # There is no revision name specified, so default
    # to the current revision
    project_open $optshash(project) -current_revision
```
If you do not require this flexibility or error checking, you can use just the `project_open` command.

**Simple Method to Open Projects**

```tcl
set proj_name [lindex $argv 0]
project_open $proj_name
```

**The quartus() Array**

The scripts in the preceding examples parsed command line arguments found in `quartus(args)`. The global `quartus()` Tcl array includes other information about your project and the current Quartus Prime executable that might be useful to your scripts. For information on the other elements of the `quartus()` array, type the following command at a Tcl prompt:

```
help -quartus
```

**The Quartus Prime Tcl Shell in Interactive Mode**

This section presents how to make project assignments and then compile the finite impulse response (FIR) filter tutorial project with the `quartus_sh` interactive shell. This example assumes that you already have the `fir_filter` tutorial design files in a project directory.

To begin, type the following at the system command prompt to run the interactive Tcl shell:

```
quartus_sh -s
```

Create a new project called `fir_filter`, with a revision called `filtref` by typing the following command at a Tcl prompt:

```
project_new -revision filtref fir_filter
```

**Note:** If the project file and project name are the same, the Quartus Prime software gives the revision the same name as the project.

Because the revision named `filtref` matches the top-level file, all design files are automatically picked up from the hierarchy tree.

Next, set a global assignment for the device with the following command:

```
set_global_assignment -name family <device family name>
```

To learn more about assignment names that you can use with the `-name` option, refer to Quartus Prime Help.

**Note:** For assignment values that contain spaces, enclose the value in quotation marks.
To compile a design, use the ::quartus::flow package, which properly exports the new project assignments and compiles the design with the proper sequence of the command-line executables. First, load the package:

    load_package flow

It returns the following:

    1.0

To perform a full compilation of the FIR filter design, use the execute_flow command with the -compile option:

    execute_flow -compile

This command compiles the FIR filter tutorial project, exporting the project assignments and running quartus_syn, quartus_fit, quartus_asm, and quartus_sta. This sequence of events is the same as selecting Start Compilation from the Processing menu in the Quartus Prime GUI.

When you are finished with a project, close it with the project_close command.

To exit the interactive Tcl shell, type exit at a Tcl prompt.

The tclsh Shell

On the UNIX and Linux operating systems, the tclsh shell included with the Quartus Prime software is initialized with a minimal PATH environment variable. As a result, system commands might not be available within the tclsh shell because certain directories are not in the PATH environment variable.

To include other directories in the path searched by the tclsh shell, set the QUARTUS_INIT_PATH environment variable before running the tclsh shell. Directories in the QUARTUS_INIT_PATH environment variable are searched by the tclsh shell when you execute a system command.

Tcl Scripting Basics

The core Tcl commands support variables, control structures, and procedures. Additionally, there are commands for accessing the file system and network sockets, and running other programs. You can create platform-independent graphical interfaces with the Tk widget set.

Tcl commands are executed immediately as they are typed in an interactive Tcl shell. You can also create scripts (including the examples in this chapter) in files and run them with the Quartus Prime executables or with the tclsh shell.

Hello World Example

The following shows the basic “Hello world” example in Tcl:

    puts "Hello world"

Use double quotation marks to group the words hello and world as one argument. Double quotation marks allow substitutions to occur in the group. Substitutions can be simple variable substitutions, or the result of running a nested command. Use curly braces {} for grouping when you want to prevent substitutions.
Variables

Assign a value to a variable with the set command. You do not have to declare a variable before using it. Tcl variable names are case-sensitive.

```
set a 1
```

To access the contents of a variable, use a dollar sign (“$”) before the variable name. The following example prints "Hello world" in a different way.

```
set a Hello
set b world
puts "$a $b"
```

Substitutions

Tcl performs three types of substitution:

- Variable value substitution
- Nested command substitution
- Backslash substitution

**Variable Value Substitution**

Variable value substitution, refers to accessing the value stored in a variable with a dollar sign (“$”) before the variable name.

**Nested Command Substitution**

Nested command substitution refers to how the Tcl interpreter evaluates Tcl code in square brackets. The Tcl interpreter evaluates nested commands, starting with the innermost nested command, and commands nested at the same level from left to right. Each nested command result is substituted in the outer command.

```
set a [string length foo]
```

**Backslash Substitution**

Backslash substitution allows you to quote reserved characters in Tcl, such as dollar signs (“$”) and braces (“{ }”). You can also specify other special ASCII characters like tabs and new lines with backslash substitutions. The backslash character is the Tcl line continuation character, used when a Tcl command wraps to more than one line.

```
set this_is_a_long_variable_name [string length "Hello \ world."
```

**Arithmetic**

Use the expr command to perform arithmetic calculations. Use curly braces (“{ }”) to group the arguments of this command for greater efficiency and numeric precision.

```
set a 5
set b [expr { $a + sqrt(2) }]
```

Tcl also supports boolean operators such as && (AND), || (OR), ! (NOT), and comparison operators such as < (less than), > (greater than), and == (equal to).
Lists

A Tcl list is a series of values. Supported list operations include creating lists, appending lists, extracting list elements, computing the length of a list, sorting a list, and more.

```tcl
set a { 1 2 3 }
```

You can use the `lindex` command to extract information at a specific index in a list. Indexes are zero-based. You can use the index `end` to specify the last element in the list, or the index `end-<n>` to count from the end of the list. For example to print the second element (at index 1) in the list stored in `a` use the following code.

```tcl
puts [lindex $a 1]
```

The `llength` command returns the length of a list.

```tcl
puts [llength $a]
```

The `lappend` command appends elements to a list. If a list does not already exist, the list you specify is created. The list variable name is not specified with a dollar sign (“$”).

```tcl
lappend a 4 5 6
```

Arrays

Arrays are similar to lists except that they use a string-based index. Tcl arrays are implemented as hash tables. You can create arrays by setting each element individually or with the `array set` command.

To set an element with an index of `Mon` to a value of `Monday` in an array called `days`, use the following command:

```tcl
set days(Mon) Monday
```

The `array set` command requires a list of index/value pairs. This example sets the array called `days`:

```tcl
array set days { Sun Sunday Mon Monday Tue Tuesday \ 
                Wed Wednesday Thu Thursday Fri Friday Sat Saturday }
```

```tcl
set day_abbreviation Mon
puts $days($day_abbreviation)
```

Use the `array names` command to get a list of all the indexes in a particular array. The index values are not returned in any specified order. The following example is one way to iterate over all the values in an array.

```tcl
foreach day [array names days] {
    puts "The abbreviation $day corresponds to the day \ 
    name $days($day)"
}
```

Arrays are a very flexible way of storing information in a Tcl script and are a good way to build complex data structures.
Control Structures

Tcl supports common control structures, including if-then-else conditions and for, foreach, and while loops. The position of the curly braces as shown in the following examples ensures the control structure commands are executed efficiently and correctly. The following example prints whether the value of variable \texttt{a} is positive, negative, or zero.

If-Then-Else Structure

```
if { $a > 0 } {
  puts "The value is positive"
} elseif { $a < 0 } {
  puts "The value is negative"
} else {
  puts "The value is zero"
}
```

The following example uses a \texttt{for} loop to print each element in a list.

For Loop

```
set a { 1 2 3 }
for { set i 0 } { $i < [llength $a] } { incr i } {
  puts "The list element at index $i is [lindex $a $i]"
}
```

The following example uses a \texttt{foreach} loop to print each element in a list.

foreach Loop

```
set a { 1 2 3 }
foreach element $a {
  puts "The list element is $element"
}
```

The following example uses a \texttt{while} loop to print each element in a list.

while Loop

```
set a { 1 2 3 }
set i 0
while { $i < [llength $a] } {
  puts "The list element at index $i is [lindex $a $i]"
  incr i
}
```

You do not have to use the \texttt{expr} command in boolean expressions in control structure commands because they invoke the \texttt{expr} command automatically.

Procedures

Use the \texttt{proc} command to define a Tcl procedure (known as a subroutine or function in other scripting and programming languages). The scope of variables in a procedure is local to the procedure. If the procedure returns a value, use the \texttt{return} command to return the value from the procedure. The following example defines a procedure that multiplies two numbers and returns the result.
Simple Procedure

```tcl
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
```

The following example shows how to use the `multiply` procedure in your code. You must define a procedure before your script calls it.

Using a Procedure

```tcl
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
set a 1
set b 2
puts [multiply $a $b]
```

Define procedures near the beginning of a script. If you want to access global variables in a procedure, use the `global` command in each procedure that uses a global variable.

Accessing Global Variables

```tcl
proc print_global_list_element { i } {
    global my_data
    puts "The list element at index $i is [lindex $my_data $i]"
}
set my_data { 1 2 3}
print_global_list_element 0
```

File I/O

Tcl includes commands to read from and write to files. You must open a file before you can read from or write to it, and close it when the read and write operations are done. To open a file, use the `open` command; to close a file, use the `close` command. When you open a file, specify its name and the mode in which to open it. If you do not specify a mode, Tcl defaults to read mode. To write to a file, specify `w` for write mode.

Open a File for Writing

```tcl
set output [open myfile.txt w]
```

Tcl supports other modes, including appending to existing files and reading from and writing to the same file.

The `open` command returns a file handle to use for read or write access. You can use the `puts` command to write to a file by specifying a filehandle.

Write to a File

```tcl
set output [open myfile.txt w]
puts $output "This text is written to the file."
close $output
```
You can read a file one line at a time with the `gets` command. The following example uses the `gets` command to read each line of the file and then prints it out with its line number.

Read from a File

```tcl
set input [open myfile.txt]
set line_num 1
while { [gets $input line] >= 0 } {
    # Process the line of text here
    puts "$line_num: $line"
    incr line_num
}
close $input
```

Syntax and Comments

Arguments to Tcl commands are separated by white space, and Tcl commands are terminated by a newline character or a semicolon. You must use backslashes when a Tcl command extends more than one line.

Tcl uses the hash or pound character (`#`) to begin comments. The `#` character must begin a comment. If you prefer to include comments on the same line as a command, be sure to terminate the command with a semicolon before the `#` character. The following example is a valid line of code that includes a `set` command and a comment.

```tcl
set a 1;# Initializes a
```

Without the semicolon, it would be an invalid command because the `set` command would not terminate until the new line after the comment.

The Tcl interpreter counts curly braces inside comments, which can lead to errors that are difficult to track down. The following example causes an error because of unbalanced curly braces.

```tcl
# if { $x > 0 } {
if { $y > 0 } {
    # code here
}
```

External References

For more information about Tcl, refer to the following sources:


Related Information

- [Quartus Prime Tcl Examples](tcl.activestate.com)
  For Quartus Prime Tcl example scripts
- [tcl.activestate.com](tcl.activestate.com)
  Tcl Developer Xchange
# Document Revision History

## Table 5-4: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2015.11.02    | 15.1.0  | • Changed instances of *Quartus II* to *Quartus Prime*.  
• Updated the list of Tcl packages in the *Quartus Prime Tcl Packages* section.  
• Updated the *Quartus Prime Tcl API Help* section:  
  • Updated the Tcl Help Output |
| June 2014     | 14.0.0  | Updated the format. |
| June 2012     | 12.0.0  | • Removed survey link. |
| November 2011 | 11.0.1  | • Template update  
• Updated supported version of Tcl in the section “Tool Command Language.”  
• minor editorial changes |
| May 2011      | 11.0.0  | Minor updates throughout document. |
| December 2010 | 10.1.0  | Template update  
Updated to remove tcl packages used by the Classic Timing Analyzer |
| July 2010     | 10.0.0  | Minor updates throughout document. |
| November 2009 | 9.1.0   | • Removed LogicLock Plus example.  
• Added the incremental_compilation, insystem_source_probe, and rtl packages to Table 3-1 and Table 3-2.  
• Added quartus_map to table 3-2. |
| March 2009    | 9.0.0   | • Removed the “EDA Tool Assignments” section  
• Added the section “Compile All Revisions” on page 3–9  
• Added the section “Using the tclsh Shell” on page 3–20 |
| November 2008 | 8.1.0   | Changed to 8½” × 11” page size. No change to content. |
| May 2008      | 8.0.0   | Updated references. |

## Related Information

**Quartus Handbook Archive**
For previous versions of the *Quartus II Handbook*, refer to the Quartus Handbook Archive.
Altera FPGAs and CPLDs offer a multitude of configurable options to allow you to implement a custom application-specific circuit on your PCB.

Your Quartus Prime project provides important information specific to your programmable logic design, which you can use in conjunction with the device literature available on Altera’s website to ensure that you implement the correct board-level connections in your schematic.

Refer to the **Settings** dialog box options, the Fitter report, and Messages window when creating and reviewing your PCB schematic. The Quartus Prime software also provides the Pin Planner and the SSN Analyzer to assist you during your PCB schematic review process.

**Related Information**

- Schematic Review Worksheets
- Pin Connection Guidelines

## Reviewing Quartus Prime Software Settings

Review these settings in the Quartus Prime software to help you review your PCB schematic.

The **Device** dialog box in the Quartus Prime software allows you to specify device-specific assignments and settings. You can use the **Device** dialog box to specify general project-wide options, including specific device and pin options, which help you to implement correct board-level connections in your PCB schematic.

The **Device** dialog box provides project-specific device information, including the target device and any migration devices you specify. Using migration devices can impact the number of available user I/O pins and internal resources, as well as require connection of some user I/O pins to power/ground pins to support migration.

If you want to use vertical migration, which allows you to use different devices with the same package, you can specify your list of migration devices in the **Migration Devices** dialog box. The Fitter places the pins in your design based on your targeted migration devices, and allows you to use only I/O pins that are common to all of the migration devices.

If a migration device has pins that are power or ground, but the pins are also user I/O pins on a different device in the migration path, the Fitter ensures that these pins are not used as user I/O pins. You must ensure that these pins are connected to the appropriate plane on the PCB.
If you are migrating from a smaller device with NC (no-connect) pins to a larger device with power or ground pins in the same package, you can safely connect the NC pins to power or ground pins to facilitate successful migration.

Related Information
Migration Devices Dialog Box
For more information about the Migration Devices dialog box in the Quartus Prime software

Device and Pins Options Dialog Box Settings
You can set device and pin options and verify important design-specific data in the Device and Pin Options dialog box, including options found on the General, Configuration, Unused Pin, Dual-Purpose Pins, and Voltage pages.

Configuration Settings
The Configuration page of the Device and Pin Options dialog box specifies the configuration scheme and configuration device for the target device. Use the Configuration page settings to verify the configuration scheme with the MSEL pin settings used on your PCB schematic and the I/O voltage of the configuration scheme.

Your specific configuration settings may impact the availability of some dual-purpose I/O pins in user mode.

Related Information
Dual-Purpose Pins Settings on page 6-2

Unused Pin Settings
The Unused Pin page specifies the behavior of all unused pins in your design. Use the Unused Pin page to ensure that unused pin settings are compatible with your PCB. For example, if you reserve all unused pins as outputs driving ground, you must ensure that you do not connect unused I/O pins to VCC pins on your PCB. Connecting unused I/O pins to VCC pins may result in contention that could lead to higher than expected current draw and possible device overstress.

The Reserve all unused pins list shows available unused pin state options for the target device. The default state for each pin is the recommended setting for each device family.

When you reserve a pin as output driving ground, the Fitter connects a ground signal to the output pin internally. You should connect the output pin to the ground plane on your PCB, although you are not required to do so. Connecting the output driving ground to the ground plane is known as creating a virtual ground pin, which helps to minimize simultaneous switching noise (SSN) and ground bounce effects.

Dual-Purpose Pins Settings
The Dual-Purpose Pins page specifies how configuration pins should be used after device configuration completes. You can set the function of the dual-purpose pins by selecting a value for a specific pin in the Dual-purpose pins list. Pin functions should match your PCB schematic. The available options on the Dual-Purpose Pins page may differ depending on the selected configuration mode.
Voltage Settings

The Voltage page specifies the default VCCIO I/O bank voltage and the default I/O bank voltage for the pins on the target device. VCCIO I/O bank voltage settings made in the Voltage page are overridden by I/O standard assignments made on I/O pins in their respective banks.

Related Information
Reviewing Device Pin-Out Information in the Fitter Report on page 6-3

Error Detection CRC Settings

The Error Detection CRC page specifies error detection cyclic redundancy check (CRC) use for the target device. When Enable error detection CRC is turned on, the device checks the validity of the programming data in the device. Any changes made in the data while the device is in operation generates an error.

Turning on the Enable open drain on CRC error pin option allows the CRC ERROR pin to be set as an open-drain pin in some devices, which decouples the voltage level of the CRC ERROR pin from VCCIO voltage. You must connect a pull-up resistor to the CRC ERROR pin on your PCB if you turn on this option.

In addition to settings in the Device dialog box, you should verify settings in the Voltage page of the Settings dialog box.

Related Information
Device and Pin Options Dialog Box
For more information about the Device and Pins Options dialog box in the Quartus Prime software

Voltage Settings

The Voltage page, under Operating Settings and Conditions in the Settings dialog box, allows you to specify voltage operating conditions for timing and power analyses. Ensure that the settings in the Voltage page match the settings in your PCB schematic, especially if the target device includes transceivers.

The Voltage page settings requirements differ depending on the settings of the transceiver instances in the design. Refer to the Fitter report for the required settings, and verify that the voltage settings are correctly set up for your PCB schematic.

After verifying your settings in the Device and Settings dialog boxes, you can verify your device pin-out with the Fitter report.

Related Information
Pin Connection Guidelines
For more information about voltage settings

Reviewing Device Pin-Out Information in the Fitter Report

After you compile your design, you can use the reports in the Resource section of the Fitter report to check your device pin-out in detail.

The Input Pins, Output Pins, and Bidirectional Pins reports identify all the user I/O pins in your design and the features enabled for each I/O pin. For example, you can find use of weak internal pull-ups, PCI
clamp diodes, and on-chip termination (OCT) pin assignments in these sections of the Fitter report. You can check the pin assignments reported in the Input Pins, Output Pins, and Bidirectional Pins reports against your PCB schematic to determine whether your PCB requires external components.

These reports also identify whether you made pin assignments or if the Fitter automatically placed the pins. If the Fitter changed your pin assignments, you should make these changes user assignments because the location of pin assignments made by the Fitter may change with subsequent compilations.

**Figure 6-1: Resource Section Report**

This figure shows the pins the Fitter chose for the OCT external calibration resistor connections (RUP/RDN) and the name of the associated termination block in the Input Pins report. You should make these types of assignments user assignments.

The I/O Bank Usage report provides a high-level overview of the VCCIO and VREF requirements for your design, based on your I/O assignments. Verify that the requirements in this report match the settings in your PCB schematic. All unused I/O banks, and all banks with I/O pins with undefined I/O standards, default the VCCIO voltage to the voltage defined in the Voltage page of the Device and Pin Options dialog box.

The All Package Pins report lists all the pins on your device, including unused pins, dedicated pins and power/ground pins. You can use this report to verify pin characteristics, such as the location, name, usage, direction, I/O standard and voltage for each pin with the pin information in your PCB schematic. In particular, you should verify the recommended voltage levels at which you connect unused dedicated inputs and I/O and power pins, especially if you selected a migration device. Use the All Package Pins report to verify that you connected all the device voltage rails to the voltages reported.
Errors commonly reported include connecting the incorrect voltage to the predriver supply (VCCPD) pin in a specific bank, or leaving dedicated clock input pins floating. Unused input pins that should be connected to ground are designated as GND+ in the Pin Name/Usage column in the All Package Pins report.

You can also use the All Package Pins report to check transceiver-specific pin connections and verify that they match the PCB schematic. Unused transceiver pins have the following requirements, based on the pin designation in the Fitter report:

- GXB_GND—Unused GXB receiver or dedicated reference clock pin. This pin must be connected to GXB_GND through a 10k Ohm resistor.
- GXB_NC—Unused GXB transmitter or dedicated clock output pin. This pin must be disconnected.

Some transceiver power supply rails have dual voltage capabilities, such as VCCA_L/R and VCCH_L/R, that depend on the settings you created for the ALTGX parameter editor. Because these user-defined settings overwrite the default settings, you should use the All Package Pins report to verify that these power pins on the device symbol in the PCB schematics are connected to the voltage required by the transceiver. An incorrect connection may cause the transceiver to function not as expected.

If your design includes a memory interface, the DQS Summary report provides an overview of each DQ pin group. You can use this report to quickly confirm that the correct DQ/DQS pins are grouped together.

Finally, the Fitter Device Options report summarizes some of the settings made in the Device and Pin Options dialog box. Verify that these settings match your PCB schematics.

### Reviewing Compilation Error and Warning Messages

If your project does not compile without error or warning messages, you should resolve the issues identified by the Compiler before signing off on your pin-out or PCB schematic. Error messages often indicate illegal or unsupported use of the device resources and IP.

Additionally, you should cross-reference fitting and timing analysis warnings with the design implementation. Timing may be constrained due to nonideal pin placement. You should investigate if you can reassign pins to different locations to prevent fitting and timing analysis warnings. Ensure that you review each warning and consider its potential impact on the design.

### Using Additional Quartus Prime Software Features

You can generate IBIS files, which contain models specific to your design and selected I/O standards and options, with the Quartus Prime software.

Because board-level simulation is important to verify, you should check for potential signal integrity issues. You can turn on the Board-Level Signal Integrity feature in the EDA Tool Settings page of the Settings dialog box.

Additionally, using advanced I/O timing allows you to enter physical PCB information to accurately model the load seen by an output pin. This feature facilitates accurate I/O timing analysis.

### Related Information

- **Signal Integrity Analysis with Third-Party Tools** on page 14-1
  For more information about signal integrity analysis in the Quartus Prime software
Using Additional Quartus Prime Software Tools

Use the Pin Planner and the SSN Analyzer to assist you with reviewing your PCB schematics.

Pin Planner

The Quartus Prime Pin Planner helps you visualize, plan, and assign device I/O pins in a graphical view of the target device package. You can quickly locate various I/O pins and assign them design elements or other properties to ensure compatibility with your PCB layout.

You can use the Pin Planner to verify the location of clock inputs, and whether they have been placed on dedicated clock input pins, which is recommended when your design uses PLLs.

You can also use the Pin Planner to verify the placement of dedicated SERDES pins. SERDES receiver inputs can be placed only on DIFFIO_RX pins, while SERDES transmitter outputs can be placed only on DIFFIO_TX pins.

The Pin Planner gives a visual indication of signal-to-signal proximity in the Pad View window, and also provides information about differential pin pair placement, such as the placement of pseudo-differential signals.

Related Information

- **I/O Management** on page 2-1
  For more information about the Pin Planner

SSN Analyzer

The SSN Analyzer supports pin planning by estimating the voltage noise caused by the simultaneous switching of output pins on the device. Because of the importance of the potential SSN performance for a specific I/O placement, you can use the SSN Analyzer to analyze the effects of aggressor I/O signals on a victim I/O pin.

Document Revision History

Table 6-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Template update.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>Minor update of Pin Planner description for task and report windows.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
</tbody>
</table>
### Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template. No change to content.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

**Related Information**

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

This chapter introduces features in the Altera Quartus Prime software that you can use to achieve the highest design performance when you design for programmable logic devices (PLDs), especially high density FPGAs.

Physical implementation can be an intimidating and challenging phase of the design process. The Quartus Prime software provides a comprehensive environment for FPGA designs, delivering unmatched performance, efficiency, and ease-of-use.

In a typical design flow, you must synthesize your design with Quartus Prime integrated synthesis or a third-party tool, place and route your design with the Fitter, and use the TimeQuest timing analyzer to ensure your design meets the timing requirements. With the PowerPlay Power Analyzer, you ensure the design’s power consumption is within limits.

Initial Compilation: Required Settings

There are basic assignments and settings Altera recommends for your initial compilation. Check the following settings before compiling your design in the Quartus Prime software. Significantly varied compilation results can occur depending on the assignments that you set.

Device Settings

Device assignments determine the timing model that the Quartus Prime software uses during compilation.

Choose the correct speed grade to obtain accurate results and the best optimization. The device size and the package determine the device pin-out and the available resources in the device.

Device Migration Settings

If you anticipate a change to the target device later in the design cycle, either because of changes in your design or other considerations, plan for the change at the beginning of your design cycle.

Whenever you select a target device, you can also list any other compatible devices you can migrate by clicking on the Migration Devices button in the Device dialog box.

Selecting the migration device and companion device early in the design cycle helps to minimize changes to your design at a later stage.
I/O Assignments

The I/O standards and drive strengths specified for a design affect I/O timing. Specify I/O assignments so that the Quartus Prime software uses accurate I/O timing delays in timing analysis and Fitter optimizations.

If there is no PCB layout requirement, then you do not need to specify pin locations. If your pin locations are not fixed due to PCB layout requirements, then leave the pin locations unconstrained. If your pin locations are already fixed, then make pin assignments to constrain the compilation appropriately.

Use the Assignment Editor and Pin Planner to assign I/O standards and pin locations.

Related Information

- **Timing Closure and Optimization** on page 9-1
  For more information about recommendations for making pin assignments that can have a large effect on your results in smaller macrocell-based architectures.

- **I/O Management** on page 2-1
  For more information about I/O standards and pin constraints, refer to the appropriate device handbook. For more information about planning and checking I/O assignments.

Timing Requirement Settings

Use your real requirements to get the best results. If you apply more demanding timing requirements than you need, then increased resource usage, higher power utilization, increased compilation time, or all of these may result.

You must use comprehensive timing requirement settings to achieve the best results for the following reasons:

- Correct timing assignments enable the software to work hardest to optimize the performance of the timing-critical parts of your design and make trade-offs for performance. This optimization can also save area or power utilization in non-critical parts of your design.
- If enabled, the Quartus Prime software performs physical synthesis optimizations based on timing requirements.
- Depending on the **Fitter Effort** setting, the Fitter can reduce runtime if your design meets the timing requirements.

The Quartus Prime TimeQuest Timing Analyzer determines if the design implementation meets the timing requirement. The Compilation Report shows whether your design meets the timing requirements, while the timing analysis reporting commands provide detailed information about the timing paths.

To create timing constraints for the TimeQuest analyzer, create a Synopsys Design Constraints File (.sdc). You can also enter constraints in the TimeQuest GUI. Use the **write_sdc** command, or the Constraints menu in the TimeQuest analyzer. Click **Write SDC File** to write your constraints to an .sdc. You can add an .sdc to your project on the **Quartus Prime Settings** page under **Timing Analysis Settings**.

**Note:** If you already have an .sdc in your project, using the **write_sdc** command from the command line or using the **Write SDC File** option from the TimeQuest GUI allows you to create a new .sdc that combines the constraints from your current .sdc and any new constraints added through the GUI or command window, or overwrites the existing .sdc with your newly applied constraints.

Ensure that every clock signal has an accurate clock setting constraint. If clocks arrive from a common oscillator, then they are related. Ensure that you set up all related or derived clocks in the constraints correctly. You must constrain all I/O pins that require I/O timing optimization. Specify both minimum and maximum timing constraints as applicable. If your design contains more than one clock or contains...
pins with different I/O requirements, make multiple clock settings and individual I/O assignments instead of using a global constraint.

Make any complex timing assignments required in your design, including false path and multicycle path assignments. Common situations for these types of assignments include reset or static control signals (when the time required for a signal to reach a destination is not important) or paths that have more than one clock cycle available for operation in a design. These assignments enable the Quartus Prime software to make appropriate trade-offs between timing paths and can enable the Compiler to improve timing performance in other parts of your design.

**Note:** To ensure that you apply constraints or assignments to all design nodes, you can report all unconstrained paths in your design with the **Report Unconstrained Paths** command in the **Task** pane of the Quartus Prime TimeQuest Timing Analyzer or the `report_ucp` Tcl command.

**Related Information**
- **Timing Closure and Optimization** on page 9-1
  For more information about optimization with physical synthesis.
- **Advanced Settings (Fitter)**
  For more information about reducing runtime by changing Fitter effort.
- **The Quartus Prime TimeQuest Timing Analyzer**
  For more information about timing assignments and timing analysis.
- **Quartus Prime TimeQuest Timing Analyzer Cookbook**
  For more information about timing assignments and timing analysis.

**Physical Implementation**

Most optimization issues involve preserving previous results, reducing area, reducing critical path delay, reducing power consumption, and reducing runtime.

The Quartus Prime software includes advisors to address each of these issues and helps you optimize your design. Run these advisors during physical implementation for advice about your specific design.

You can reduce the time spent on design iterations by following the recommended design practices for designing with Altera® devices. Design planning is critical for successful design timing implementation and closure.

**Related Information**
- **Design Planning with the Quartus Prime Software**

**Trade-Offs and Limitations**

Many optimization goals can conflict with one another, so you might need to resolve conflicting goals. For example, one major trade-off during physical implementation is between resource usage and critical path timing, because certain techniques (such as logic duplication) can improve timing performance at the cost of increased area. Similarly, a change in power requirements can result in area and timing trade-offs, such as if you reduce the number of available high-speed tiles, or if you attempt to shorten high-power nets at the expense of critical path nets.

In addition, system cost and time-to-market considerations can affect the choice of device. For example, a device with a higher speed grade or more clock networks can facilitate timing closure at the expense of higher power consumption and system cost.
Finally, not all designs can be realized in a hardware circuit with limited resources and given constraints. If you encounter resource limitations, timing constraints, or power constraints that cannot be resolved by the Fitter, consider rewriting parts of the HDL code.

Related Information

- Timing Closure and Optimization on page 9-1

Reducing Area

By default, the Quartus Prime Fitter might physically spread a design over the entire device to meet the set timing constraints. If you prefer to optimize your design to use the smallest area, you can change this behavior. If you require reduced area, you can enable certain physical synthesis options to modify your netlist to create a more area-efficient implementation, but at the cost of increased runtime and decreased performance.

Related Information

- Netlist Optimizations and Physical Synthesis on page 13-1
- Timing Closure and Optimization on page 9-1

Recommended HDL Coding Styles

Reducing Critical Path Delay

To meet complex timing requirements involving multiple clocks, routing resources, and area constraints, the Quartus Prime software offers a close interaction between synthesis, timing analysis, floorplan editing, and place-and-route processes.

By default, the Quartus Prime Fitter tries to meet the specified timing requirements and stops trying when the requirements are met. Therefore, using realistic constraints is important to successfully close timing. If you under-constrain your design, you may get sub-optimal results. By contrast, if you over-constrain your design, the Fitter might over-optimize non-critical paths at the expense of true critical paths. In addition, you might incur an increased area penalty. Compilation time may also increase because of excessively tight constraints.

If your resource usage is very high, the Quartus Prime Fitter might have trouble finding a legal placement. In such circumstances, the Fitter automatically modifies some of its settings to try to trade off performance for area.

The Quartus Prime Fitter offers a number of advanced options that can help you improve the performance of your design when you properly set constraints. Use the Timing Optimization Advisor to determine which options are best suited for your design.

In high-density FPGAs, routing accounts for a major part of critical path timing. Because of this, duplicating or retiming logic can allow the Fitter to reduce delay on critical paths. The Quartus Prime software offers push-button netlist optimizations and physical synthesis options that can improve design performance at the expense of considerable increases of compilation time and area. Turn on only those options that help you keep reasonable compilation times and resource usage. Alternately, you can modify your HDL to manually duplicate or adjust the timing logic.

Reducing Power Consumption

The Quartus Prime software has features that help reduce design power consumption. The PowerPlay power optimization options control the power-driven compilation settings for Synthesis and the Fitter.
Reducing Runtime

Many Fitter settings influence compilation time. Most of the default settings in the Quartus Prime software are set for reduced compilation time. You can modify these settings based on your project requirements.

The Quartus Prime software supports parallel compilation in computers with multiple processors. This can reduce compilation times by up to 15% while giving the identical result as serial compilation.

Using Quartus Prime Tools

The following sections describe several Quartus Prime tools that you can use to help optimize your design.

Design Analysis

The Quartus Prime software provides tools that help with a visual representation of your design. You can use the RTL Viewer to see a schematic representation of your design before synthesis and place-and-route. The Technology Map Viewer provides a schematic representation of the design implementation in the selected device architecture after synthesis and place-and-route. It can also include timing information.

Advisors

The Quartus Prime software includes several advisors to help you optimize your design and reduce compilation time.

You can complete your design faster by following the recommendations in the following advisor. These advisors give recommendations based on your project settings and your design constraints:

- Resource Optimization Advisor
- Timing Optimization Advisor
- Power Optimization Advisor
- Compilation Time Advisor
- Pin Optimization Advisor
- Arria 10 to Stratix 10 Migration Advisor

Design Space Explorer II

Use Design Space Explorer II (DSE) to find optimal settings in the Quartus Prime software.

DSE II automatically tries different combinations of netlist optimizations and advanced Quartus Prime software compiler settings, and reports the best settings for your design, based on your chosen primary optimization goal. You can try different seeds with DSE II if you are fairly close to meeting your timing or area requirements and find one seed that meets timing or area requirements. Finally, DSE II can run compilations on a remote compute farm, which shortens the timing closure process.

Related Information

About Design Space Explorer II
# Document Revision History

## Table 7-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated DSE II content.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Minor changes for HardCopy.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>Added the information about initial compilation requirements. This section was moved from the Area Optimization chapter of the Quartus Prime Handbook. Minor updates to delineate division of Timing and Area optimization chapters.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.3</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.2</td>
<td>Changed to new document template. No change to content.</td>
</tr>
<tr>
<td>August 2010</td>
<td>10.0.1</td>
<td>Corrected link</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release. Chapter based on topics and text in Section III of volume 2.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

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

The Analysis and Synthesis and Fitter modules consume the majority of time in a compilation. The Quartus Prime software offers several features and techniques to help reduce compilation time.

The Analysis and Synthesis module includes physical synthesis optimizations performed during synthesis, if you have turned on physical synthesis optimizations. The Fitter includes two steps, placement and routing, and also includes physical synthesis if you turned on the physical synthesis option with Normal or Extra effort levels. The Flow Elapsed Time section of the Compilation Report shows the duration of the Analysis and Synthesis and Fitter modules. The Fitter Messages report in the Fitter section of the Compilation Report displays the elapsed time for placement and routing processes.

Placement is the process of finding optimum locations for the logic in your design. Placement includes Quartus Prime pre-Fitter operations, which place dedicated logic such as clocks, PLLs, and transceiver blocks. Routing is the process of connecting the nets between the logic in your design. Finding better placement for the logic in your design requires more compilation time. Good logic placement allows you to more easily meet your timing requirements and makes your design easier to route.

Info: Fitter placement operations ending: elapsed time = <days:hours:minutes:seconds>
Info: Fitter routing operations ending: elapsed time = <days:hours:minutes:seconds>

The Quartus Prime software displays info messages while the Fitter is running (including Placement and Routing). The Message window displays this message every hour to indicate Fitter operations are progressing normally.

Info: Placement optimizations have been running for 4 hour(s)

Compilation Time Advisor

A Compilation Time Advisor is available to help you to reduce compilation time. Run the Compilation Time Advisor on the Tools menu by pointing to Advisors and clicking Compilation Time Advisor. You can find all the compilation time optimizing techniques described in this section in the Compilation Time Advisor as well.
Strategies to Reduce the Overall Compilation Time

You can use the following strategies to reduce the overall time required to compile your design:

- Parallel compilation (for systems with multiple processor cores)
- Rapid Recompile and Smart Compilation reuse results from a previous compilation to reduce overall 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 8-1: 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.
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.
Reducing Synthesis Time and Synthesis Netlist Optimization Time

You can reduce synthesis time without affecting the Fitter time by reducing your use of netlist optimizations. For tips on reducing synthesis time when using third-party EDA synthesis tools, refer to your synthesis software’s documentation.

Settings to Reduce Synthesis Time and Synthesis Netlist Optimization Time

You can use Quartus Prime integrated synthesis to synthesize and optimize HDL designs, and you can use synthesis netlist optimizations to optimize netlists that were synthesized by third-party EDA software. When using Quartus Prime Integrated Synthesis, you can also enable Physical Synthesis Optimization before performing Analysis and Synthesis. Netlist optimizations can cause the Analysis and Synthesis module to take much longer to run. Read the Analysis and Synthesis messages to determine how much
time these optimizations take. The compilation time spent in Analysis and Synthesis is usually short compared to the compilation time spent in the Fitter.

If your design meets your performance requirements without synthesis netlist optimizations, turn off the optimizations to save time. If you require synthesis netlist optimizations to meet performance, you can optimize parts of your design hierarchy separately to reduce the overall time spent in Analysis and Synthesis.

Turn off settings that are not useful. In general, if you carry over compilation settings from a previous project, evaluate all settings and keep only those that you need.

Use Appropriate Coding Style to Reduce Synthesis Time  
Your HDL coding style can also affect the synthesis time. For example, if you want to infer RAM blocks from your code, you must follow the guidelines for inferring RAMs. If RAM blocks are not inferred properly, the software implements those blocks as registers.

If you are trying to infer a large memory block, the software consumes more resources in the FPGA. This can cause routing congestion and increasing compilation time significantly. If you see high routing utilizations in certain blocks, it is a good idea to review the code for such blocks.

Related Information

Recommended HDL Coding Styles
For more information about coding guidelines.

Reducing Placement Time

The time required to place a design depends on two factors: the number of ways the logic in your design can be placed in the device and the settings that control the amount of effort required to find a good placement. You can reduce the placement time in two ways:

- Change the settings for the placement algorithm.

Sometimes there is a trade-off between placement time and routing time. Routing time can increase if the placer does not run long enough to find a good placement. When you reduce placement time, ensure that it does not increase routing time and negate the overall time reduction.

Fitter Effort Setting

The highest Fitter effort setting, Standard Fit, requires the most runtime, but does not always yield a better result than using the default Auto Fit.

For designs with very tight timing requirements, both Auto Fit and Standard Fit use the maximum effort during optimization. Altera recommends using Auto Fit for reducing compilation time. If you are certain that your design has only easy-to-meet timing constraints, you can select Fast Fit for an even greater runtime savings.

Placement Effort Multiplier Settings

You can control the amount of time the Fitter spends in placement by reducing with the Placement Effort Multiplier option.

Click Assignments > Settings > Compiler Settings > Advanced Settings (Fitter) and specify a value for Placement Effort Multiplier. The default is 1.0. Legal values must be greater than 0 and can be non-integer values. Numbers between 0 and 1 can reduce fitting time, but also can reduce placement quality and design performance.
Physical Synthesis Effort Settings

Physical synthesis options enable you to optimize your post-synthesis netlist and improve your timing performance. These options, which affect placement, can significantly increase compilation time. If your design meets your performance requirements without physical synthesis options, turn them off to reduce compilation time. For example, if some or all of the physical synthesis algorithm information messages display an improvement of 0 ps, turning off physical synthesis can reduce compilation time. You also can use the Physical synthesis effort setting on the Advanced Fitter Settings dialog box to reduce the amount of extra compilation time used by these optimizations.

The Fast setting directs the Quartus Prime software to use a lower level of physical synthesis optimization. Compared to the Normal physical synthesis effort level, using the Fast setting can cause a smaller increase in compilation time. However, the lower level of optimization can result in a smaller increase in design performance.

Reducing Routing Time

The time required to route a design depends on three factors: the device architecture, the placement of your design in the device, and the connectivity between different parts of your design. The routing time is usually not a significant amount of the compilation time. If your design requires a long time to route, perform one or more of the following actions:

- Check for routing congestion.
- Turn off Fitter Aggressive Routability Optimization.

Identifying Routing Congestion in the Chip Planner

To identify areas of routing congestion in your design, open the Chip Planner from the Tools menu. To view the routing congestion in the Chip Planner, double-click the Report Routing Utilization command in the Tasks list. Click Preview in the Report Routing Utilization dialog box to preview the default congestion display. Change the Routing utilization type to display congestion for specific resources. The default display uses dark blue for 0% congestion and red for 100%. Adjust the slider for Threshold percentage to change the congestion threshold level.

Even if average congestion is not very high, your design may have areas where congestion is very high in a specific type of routing. You can use the Chip Planner to identify areas of high congestion for specific interconnect types. You can change the connections in your design to reduce routing congestion. If the area with routing congestion is in a LogicLock Plus region or between LogicLock Plus regions, change or remove the LogicLock Plus regions and recompile your design. If the routing time remains the same, the time is a characteristic of your design and the placement. If the routing time decreases, consider changing the size, location, or contents of LogicLock Plus regions to reduce congestion and decrease routing time.

Sometimes, routing congestion may be a result of the HDL coding style used in your design. After you identity congested areas using the Chip Planner, review the HDL code for the blocks placed in those areas to determine whether you can reduce interconnect usage by code changes.

The Quartus Prime compilation messages contain information about average and peak interconnect usage. Peak interconnect usage over 75%, or average interconnect usage over 60%, could be an indication that it might be difficult to fit your design. Similarly, peak interconnect usage over 90%, or average interconnect usage over 75%, are likely to have increased chances of not getting a valid fit.

Related Information

Using Incremental Compilation
Reducing Static Timing Analysis Time

If you are performing timing-driven synthesis, the Quartus Prime software runs the TimeQuest analyzer during Analysis and Synthesis.

The Quartus Prime Fitter also runs the TimeQuest analyzer during placement and routing. If there are incorrect constraints in the Synopsys Design Constraints File (.sdc), the Quartus Prime software may spend unnecessary time processing constraints several times.

- If you do not specify false paths and multicycle paths in your design, the TimeQuest analyzer may analyze paths that are not relevant to your design.
- If you redefine constraints in the .sdc files, the TimeQuest analyzer may spend additional time processing them. To avoid this situation, look for indications that Synopsis design constraints are being redefined in the compilation messages, and update the .sdc file.
- Ensure that you provide the correct timing constraints to your design, because the software cannot assume design intent, such as which paths to consider as false paths or multicycle paths. When you specify these assignments correctly, the TimeQuest analyzer skips analysis for those paths, and the Fitter does not spend additional time optimizing those paths.

Setting Process Priority

It might be necessary to reduce the computing resources allocated to the compilation at the expense of increased compilation time. It can be convenient to reduce the resource allocation to the compilation with single processor machines if you must run other tasks at the same time.

Related Information

Processing Page (Options Dialog Box)
For more information about setting process priority, refer to Quartus Prime Help.

Document Revision History

Table 8-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>• Updated location of Fitter Settings, Analysis &amp; Synthesis Settings,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Physical Synthesis Optimizations to Compiler Settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about Rapid Recompile feature.</td>
</tr>
<tr>
<td>2014.08.18</td>
<td>14.0a10.0</td>
<td>Added restriction about smart compilation in Arria 10 devices.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
Updated “Placement Effort Multiplier Settings” section.  
Updated “Identifying Routing Congestion in the Chip Planner” section.  
General editorial changes throughout the chapter. |
| June 2012  | 12.0.0  | Removed survey link.                                                   |
| November 2011 | 11.0.1 | Template update.                                                      |
| May 2011   | 11.0.0  | Updated “Using Parallel Compilation with Multiple Processors”.  
Updated “Identifying Routing Congestion in the Chip Planner”.  
General editorial changes throughout the chapter. |
| December 2010 | 10.1.0 | Template update.  
Added details about peak and average interconnect usage.  
Added new section “Reducing Static Timing Analysis Time”.  
Minor changes throughout chapter. |
| July 2010  | 10.0.0  | Initial release.                                                       |

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
About Timing Closure and Optimization

This manual describes techniques to improve timing performance when designing for Altera® devices.

The application techniques vary between designs. Applying each technique does not always improve results. Settings and options in the Quartus Prime software have default values that provide the best trade-off between compilation time, resource utilization, and timing performance. You can adjust these settings to determine whether other settings provide better results for your design.

Initial Compilation: Optional Fitter Settings

The Fitter offers many optional settings; however, this section focuses on the optional timing-optimization related Fitter settings only, which are the Optimize Hold Timing, Optimize Multi-Corner Timing, and Fitter Aggressive Routability Optimization settings.

Caution: The settings required to optimize different designs could be different. The group of settings that work best for one design may not produce the best result for another design.

Related Information
Advanced Fitter Setting Dialog Box online help
For scripting and device family support information of the Optimize Hold Timing and Optimize Multi-Corner Timing settings

Optimize Hold Timing

The Optimize Hold Timing option directs the Quartus Prime software to optimize minimum delay timing constraints. By default, the Quartus Prime software optimizes hold timing for all paths for designs for supported devices. By default, the Quartus Prime software optimizes hold timing only for I/O paths and minimum tPD paths for older devices.

When you turn on Optimize Hold Timing in the Advanced Fitter Settings dialog box, the Quartus Prime software adds delay to paths to ensure that your design meets the minimum delay requirements. If you select I/O Paths and Minimum TPD Paths, the Fitter works to meet the following criteria:

- Hold times (tH) from the device input pins to the registers
- Minimum delays from I/O pins to I/O registers or from I/O registers to I/O pins
- Minimum clock-to-out time (tCO) from registers to output pins
If you select **All Paths**, the Fitter also works to meet hold requirements from registers to registers, as highlighted in blue in the figure, in which a derived clock generated with logic causes a hold time problem on another register.

**Figure 9-1: Optimize Hold Timing Option Fixing an Internal Hold Time Violation**

However, if your design still has internal hold time violations between registers, correct the violations by manually adding some delays by instantiating LCELL primitives, or by making changes to your design, such as using a clock enable signal instead of a derived or gated clock.

**Related Information**

[Recommended Design Practices documentation](#)

For design practices that help to eliminate internal hold time violations

**Optimize Multi-Corner Timing**

Due to process variations and changes in operating conditions, delays on some paths can be significantly smaller than those in the slow corner timing model. This can result in hold time violations on those paths, and in rare cases, additional setup time violations.

Also, because of the small process geometries of newer device families, the slowest circuit performance of designs targeting these devices does not necessarily occur at the highest operating temperature. The temperature at which the circuit is slowest depends on the selected device, the design, and the compilation results. Therefore, the Quartus Prime software provides newer device families with three different timing corners—Slow 85°C corner, Slow 0°C corner, and Fast 0°C corner. For other device families, two timing corners are available—Fast 0°C and Slow 85°C corner.

The **Optimize multi-corner timing** option 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.

The **Optimize multi-corner timing** option helps to create a design implementation that is more robust across process, temperature, and voltage variations. Turning on this option increases compilation time by approximately 10%.
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).

**Fitter Aggressive Routability Optimization**

The Fitter Aggressive Routability Optimizations logic option allows you to specify 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.

This option is useful if routing resources are resulting in no-fit errors, and you want to reduce routing wire use.

The table lists the settings for the Fitter Aggressive Routability Optimizations logic option.

**Table 9-1: Fitter Aggressive Routability Optimizations Logic Option Settings**

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>The Fitter always performs aggressive routability optimizations. If you set the Fitter Aggressive Routability Optimizations logic option to Always, reducing wire utilization may affect the performance of your design.</td>
</tr>
<tr>
<td>Never</td>
<td>The Fitter never performs aggressive routability optimizations. If improving timing is more important than reducing wire usage, then set this option to Automatically or Never.</td>
</tr>
<tr>
<td>Automatically</td>
<td>The Fitter performs aggressive routability optimizations automatically, based on the routability and timing requirements of the design. If improving timing is more important than reducing wire usage, then set this option to Automatically or Never.</td>
</tr>
</tbody>
</table>

**Design Analysis**

The initial compilation establishes whether the design achieves a successful fit and meets the specified timing requirements. This section describes how to analyze your design results in the Quartus Prime software.

**Ignored Timing Constraints**

The Quartus Prime software ignores illegal, obsolete, and conflicting constraints.

You can view a list of ignored constraints by clicking Report Ignored Constraints in the Reports menu in the TimeQuest GUI or by typing the following command to generate a list of ignored timing constraints:

```
report_sdc -ignored -panel_name "Ignored Constraints"
```

You should analyze any constraints that the Quartus Prime software ignores. If necessary, correct the constraints and recompile your design before proceeding with design optimization.

You can view a list of ignored assignment in the Ignored Assignment Report generated by the Fitter.
I/O Timing (Including \( t_{PD} \))

TimeQuest analyzer supports the Synopsys Design Constraints (SDC) format for constraining your design. When using the TimeQuest analyzer for timing analysis, use the `set_input_delay` constraint to specify the data arrival time at an input port with respect to a given clock. For output ports, use the `set_output_delay` command to specify the data arrival time at an output port’s receiver with respect to a given clock. You can use the `report_timing` Tcl command to generate the I/O timing reports.

The I/O paths that do not meet the required timing performance are reported as having negative slack and are highlighted in red in the TimeQuest analyzer `Report` pane. In cases where you do not apply an explicit I/O timing constraint to an I/O pin, the Quartus Prime timing analysis software still reports the `Actual` number, which is the timing number that must be met for that timing parameter when the device runs in your system.

Related Information

- Quartus Prime TimeQuest Timing Analyzer documentation
  For more information about the `report_sdc` command and its options
- Fitter Summary Reports online help

Register-to-Register Timing

Timing Analysis with the TimeQuest Timing Analyzer

Analyze all valid register-to-register paths by using the appropriate constraints in the TimeQuest analyzer. To view all timing summaries, run the `Report All Summaries` command by double-clicking `Report All Summaries` in the Tasks pane in the TimeQuest analyzer.

If any clock domains have failing paths (highlighted in red in the Report panel), right-click the Clock Name listed in the Clocks Summary panel and go to `Report Timing` to get more details. Your design meets timing requirements when you do not have negative slack on any register-to-register path on any of the clock domains.

When timing requirements are not met, a report on the failed paths (highlighted in red) can uncover more detail.

When you select a path listed in the TimeQuest `Report Timing` pane, the tabs in the corresponding path detail pane show a path summary of source and destination registers and their timing, statistics about the path delay, detailed information about the complete data path with all nodes in the path, and the waveforms of the relevant signals. The Extra Fitter Information tab will show a Graphical Data Path of where the offending path lies on the physical device. This can reveal whether the timing failure may be distance related, due to the source and destination node being too close or too far. The Chip Planner can also be used to investigate the physical layout of a failing path in more detail. To locate a selected path in the Chip Planner, right-click a node, point to Locate, and select Locate in Chip Planner. The Chip Planner appears with the path highlighted. Use this to show fanout, fanin, routing congestion, and region assignments information, and to determine whether those factors might be contributing to the timing critical path. Additionally, if you know that a path is not a valid path, you can set it to be a false path using the shortcut menu.
The **Data Path** tab can also be useful for determining contributions to timing critical paths. The **Data Path** tab shows details of the paths that the clock and data took to get from source to destination nodes, and the time it took on an incremental and cumulative basis. It also provides information about the routing types and elements used, and their locations.

To view the path details of any selected path, click the **Data Path** tab in the path details pane. The **Data Path** tab displays the details of the Data Arrival Path, as well as the Data Required Path.

The **Waveform** tab will show the slack relationship between arrival data and required data. This could be useful for determining how close or far off the path is from meeting timing.

To aid in timing debug, the RTL Viewer or Technology Map Viewer allow you to see schematic representations of your design. These viewers allow you to view a gate-level or technology-mapped representation of your design netlist. By providing a view of the path from source and destination nodes, the viewers can help identify areas in a design that may benefit from reducing the number of logic levels between the nodes. To locate a timing path in one of the viewers, right-click a path in the report, point to **Locate**, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**.

**Related Information**

- [Quartus Prime TimeQuest Timing Analyzer documentation](#)
  
  Information about how timing analysis results are calculated

- [Analyzing Designs with Quartus Prime Netlist Viewers documentation](#)

**Tips for Analyzing Failing Paths**

When you are analyzing failing paths, examine the reports and waveforms to determine if the correct constraints are being applied, and add timing exceptions as appropriate. A multicycle constraint relaxes setup or hold relationships by the specified number of clock cycles. A false path constraint specifies paths that can be ignored during timing analysis. Both constraints allow the Fitter to work harder on affected paths.

Focus on improving the paths that show the worst slack. The Fitter works hardest on paths with the worst slack. If you fix these paths, the Fitter might be able to improve the other failing timing paths in the design.

Check for particular nodes that appear in many failing paths. These nodes will appear in a timing report panel at the top of the list, along with their minimum slacks. Look for paths that have common source registers, destination registers, or common intermediate combinational nodes. In some cases, the registers might not be identical, but are part of the same bus.

In the timing analysis report panels, clicking on the **From** or **To** column headings can help to sort the paths by the source or destination registers. Clicking first on **From**, then on **To**, uses the registers in the **To** column as the primary sort and the registers in the **From** column as the secondary sort. If you see common nodes, these nodes indicate areas of your design that might be improved through source code changes or Quartus Prime optimization settings. Constraining the placement for just one of the paths might decrease the timing performance for other paths by moving the common node further away in the device.

**Related Information**

[Design Evaluation for Timing Closure](#) on page 9-26
Tips for Analyzing Failing Clock Paths that Cross Clock Domains

When analyzing clock path failures, check whether these paths cross two clock domains. This is the case if the From Clock and To Clock in the timing analysis report are different.

Figure 9-2: Different Value in From Clock and To Clock Field

There can also be paths that involve a different clock in the middle of the path, even if the source and destination register clock are the same.

When you run Report Timing on your design, the report shows the launch clock and latch clock for each failing path. Check whether these failing paths between these clock domains should be analyzed synchronously. If the failing paths are not to be analyzed synchronously, they must be set as false paths. Also check the relationship between the launch clock and latch clock to make sure it is realistic and what you expect from your knowledge of the design. For example, the path can start at a rising edge and end at a falling edge, which reduces the setup relationship by one half clock cycle.

Review the clock skew reported in the Timing Report. A large skew may indicate a problem in your design, such as a gated clock or a problem in the physical layout (for example, a clock using local routing instead of dedicated clock routing). When you have made sure the paths are analyzed synchronously and that there is no large skew on the path, and that the constraints are correct, you can analyze the data path. These steps help you fine tune your constraints for paths across clock domains to ensure you get an accurate timing report.

Check if the PLL phase shift is reducing the setup requirement. You might be able to adjust this using PLL parameters and settings.

Paths that cross clock domains are generally protected with synchronization logic (for example, FIFOs or double-data synchronization registers) to allow asynchronous interaction between the two clock domains. In such cases, you can ignore the timing paths between registers in the two clock domains while running timing analysis, even if the clocks are related.

The Fitter attempts to optimize all failing timing paths. If there are paths that can be ignored for optimization and timing analysis, but the paths do not have constraints that instruct the Fitter to ignore them, the Fitter tries to optimize those paths as well. In some cases, optimizing unnecessary paths can prevent the Fitter from meeting the timing requirements on timing paths that are critical to the design. It is beneficial to specify all paths that can be ignored by setting false path constraints on them, so that the Fitter can put more effort into the paths that must meet their timing requirements instead of optimizing paths that can be ignored.

Related Information

Quartus Prime TimeQuest Timing Analyzer
Details about how to ignore timing paths that cross clock domains
Tips for Analyzing Paths from/to the Source and Destination of Critical Path

When analyzing the failing paths in a design, it is often helpful to get a fuller picture of the many interactions the fitter may be working on around the paths. To understand what may be pulling on a critical path, the following `report_timing` command can be useful.

In the project directory, run the `report_timing` command, shown in the example below, in a `.tcl` file to analyze the nodes in a critical path.

```
set wrst_src <insert_source_of_worst_path_here>
set wrst_dst <insert_destination_of_worst_path_here>
report_timing -setup -npaths 50 -detail path_only -from $wrst_src \
-panel_name "Worst Path||wrst_src -> *"
report_timing -setup -npaths 50 -detail path_only -to $wrst_dst \
-panel_name "Worst Path||* -> wrst_dst"
report_timing -setup -npaths 50 -detail path_only -to $wrst_src \
-panel_name "Worst Path||* -> wrst_src"
report_timing -setup -npaths 50 -detail path_only -from $wrst_dst \
-panel_name "Worst Path||wrst_dst -> *"
```

Copy the node names from the **From Node** and **To Node** columns of the worst path into the first two variables, and then in the TimeQuest timing analyzer, in the Script menu, source the `.tcl` script.

In the resulting timing panel, timing failed paths (highlighted in red) can be located in the Chip Planner, where information such as distance between the nodes and large fanouts can be viewed.

The figure shows a simplified example of what these reports analyzed.

**Figure 9-3: Timing Report**

The critical path of the design is in red. The script analyzes the path between the worst source and destination registers. The first `report_timing` command analyzes other path that the source is driving, as shown in green. The second `report_timing` command analyzes the critical path and other path going to the destination, shown in yellow. These commands report everything inside these two endpoints that are pulling them in different directions. The last two `report_timing` commands show everything outside of
the endpoints pulling them in other directions. If any of these reports have slacks near the critical path, then the Fitter is balancing these paths with the critical path, trying to achieve the best slack. The figure is quite simple compared to the critical path in most designs, but it is easy to see how this can get very complicated quickly.

**Tips for Locating Multiple Paths to the Chip Planner**

The Chip Planner can be used as a visual aid in locating timing critical paths. To view these paths from timing reports, do the following:

1. Run `report_timing` to show multiple paths.
2. Select multiple rows of the timing report.
3. Right-click, select **Locate Path**, and then click **Chip Planner**.
4. The **Locate History** window in the Chip Planner displays the selected paths and the worst path.
5. Double-click **Locate Paths** to show all paths at once, or select individual paths to view the path in the Chip Planner.

This will show whether timing failures may be due to large distances between the nodes or large fanouts.

**Tips for Creating a .tcl Script to Monitor Critical Paths Across Compiles**

Many designs have the same critical paths show up after each compile, but some suffer from having critical paths bounce around between different hierarchies, changing with each compile.

This could happen in high speed designs where many register to register paths have very little slack. Different placements can then result in timing failures in the marginal paths. In designs like this, create a `TQ_critical_paths.tcl` script in the project directory. For a given compile, view the critical paths and then write a generic `report_timing` command to capture those paths. For example, if several paths fail in a low-level hierarchy, you can add the following command:

```
report_timing -setup -npaths 50 -detail path_only \
-to "main_system: main_system_inst|app_cpu:cpu|" -\n-panel_name "Critical Paths||s: * -> app_cpu"
```

If there is a specific path, such as a bit of a state-machine going to other *count_sync* registers, you can add a command as shown by the following:

```
report_timing -setup -npaths 50 -detail path_only \ 
-from "main_system: main_system_inst|egress_count_sm:egress_inst|update" \ 
-to "count_sync:" -panel_name "Critical Paths||s: egress_sm|update -> count_sync"
```

This file can be sourced in the TimeQuest timing analyzer after every compilation, and new `report_timing` commands can be added as new critical paths appear. This helps you monitor paths that consistently fail and paths that are only marginal, so you can prioritize effectively.

**Global Routing Resources**

Global routing resources are designed to distribute high fan-out, low-skew signals (such as clocks) without consuming regular routing resources. Depending on the device, these resources can span the entire chip, or some smaller portion, such as a quadrant. The Quartus Prime software attempts to assign signals to global routing resources automatically, but you might be able to make more suitable assignments manually.
For details about the number and types of global routing resources available, refer to the relevant device handbook.

Check the global signal utilization in your design to ensure that the appropriate signals have been placed on the global routing resources. In the Compilation Report, open the Fitter report and click Resource Section. Analyze the Global & Other Fast Signals and Non-Global High Fan-out Signals reports to determine whether any changes are required.

You might be able to reduce skew for high fan-out signals by placing them on global routing resources. Conversely, you can reduce the insertion delay of low fan-out signals by removing them from global routing resources. Doing so can improve clock enable timing and control signal recovery/removal timing, but increases clock skew. Use the Global Signal setting in the Assignment Editor to control global routing resources.

**Optimizing Timing (LUT-Based Devices)**

You can use the following guidelines if your design does not meet its timing requirements:

### Debugging Timing Failures in the TimeQuest Analyzer

A **Report Timing Closure Recommendations** task is available in the Custom Reports section of the Tasks pane of the TimeQuest analyzer. Use this report to get more information and help on the failing paths in your design.

When you run the **Report Timing Closure Recommendations** task, you get specific recommendations about failing paths in your design and changes that you can make to potentially fix the failing paths.

Selecting the **Report Timing Closure Recommendations** task opens the **Report Timing Closure Recommendations** dialog box.

From the **Report Timing Closure Recommendations** dialog box, you can select paths based on the clock domain, filter by nodes on path, and choose the number of paths to analyze.

After running the **Report Timing Closure Recommendations** task in the TimeQuest analyzer, examine the reports in the **Report Timing Closure Recommendations** folder in the Report pane of the TimeQuest analyzer GUI. Each recommendation has star symbols (*) associated with it. Recommendations with more stars are more likely to help you close timing on your design.

The reports give you the most probable causes of failure for each path being analyzed. The reports are organized into sections, depending on the type of issues found in the design, such as large clock skew, restricted optimizations, unbalanced logic, skipped optimizations, coding style that has too many levels of logic between registers, or region or partition constraints specific to your project.

You will see recommendations that may help you fix the failing paths. For detailed analysis of the critical paths, run the report_timing command on specified paths. In the Extra Fitter Information tab of the Path report panel, you will also see detailed Fitter-related information that may help you visualize the issue and take the appropriate action if your constraints cause a specific placement.

### Related Information

- **Report Timing Closure Recommendations Dialog Box online help**
Timing Optimization Advisor

While the TimeQuest Report Timing Closure Recommendations task gives specific recommendations to fix failing paths, the Timing Optimization Advisor gives more general recommendations to improve timing performance for a design.

The Timing Optimization Advisor guides you in making settings that optimize your design to meet your timing requirements. To run the Timing Optimization Advisor, on the Tools menu, point to Advisors and click Timing Optimization Advisor. This advisor describes many of the suggestions made in this section.

When you open the Timing Optimization Advisor after compilation, you can find recommendations to improve the timing performance of your design. Some of the recommendations in these advisors can contradict each other. Altera recommends evaluating these options and choosing the settings that best suit the given requirements.

The example shows the Timing Optimization Advisor after compiling a design that meets its frequency requirements, but requires setting changes to improve the timing.

Figure 9-4: Timing Optimization Advisor

When you expand one of the categories in the Timing Optimization Advisor, such as Maximum Frequency (fmax) or I/O Timing (tsu, tco, tpd), the recommendations are divided into stages. The stages show the order in which to apply the recommended settings. The first stage contains the options that are easiest to change, make the least drastic changes to your design optimization, and have the least effect on compilation time. Icons indicate whether each recommended setting has been made in the current project. In the figure, the checkmark icons in the list of recommendations for Stage 1 indicate recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icons indicate general suggestions. For these entries, the advisor...
does not report whether these recommendations were followed, but instead explains how you can achieve better performance. For a legend that provides more information for each icon, refer to the “How to use” page in the Timing Optimization Advisor.

There is a link from each recommendation to the appropriate location in the Quartus Prime GUI where you can change the settings. For example, consider the Synthesis Netlist Optimizations page of the Settings dialog box or the Global Signals category in the Assignment Editor. This approach provides the most control over which settings are made and helps you learn about the settings in the software. In some cases, you can also use the Correct the Settings button to automatically make the suggested change to global settings.

For some entries in the Timing Optimization Advisor, a button appears that allows you to further analyze your design and gives you more information. The advisor provides a table with the clocks in the design and indicates whether they have been assigned a timing constraint.

I/O Timing Optimization

This stage of design optimization focuses on I/O timing. Ensure that you have made the appropriate assignments described in the “Initial Compilation: Required Settings” section in the Design Optimization Overview chapter of the Quartus Prime Handbook. You must also ensure that resource utilization is satisfactory before proceeding with I/O timing optimization. The suggestions provided in this section are applicable to all Altera FPGA families and to the MAX II family of CPLDs.

Because changes to the I/O paths affect the internal register-to-register timing, complete this stage before proceeding to the register-to-register timing optimization stage as described in Register-to-Register Timing Optimization Techniques (LUT-Based Devices).

The options presented in this section address how to improve I/O timing, including the setup delay ($t_{SU}$), hold time ($t_H$), and clock-to-output ($t_{CO}$) parameters.

Improving Setup and Clock-to-Output Times Summary

The table lists the recommended order in which to use techniques to reduce $t_{SU}$ and $t_{CO}$ times. “Yes” indicates which timing parameters are affected by each technique. Reducing $t_{SU}$ times increases hold ($t_H$) times.

Table 9-2: Improving Setup and Clock-to-Output Times

<table>
<thead>
<tr>
<th>Technique</th>
<th>Affects $t_{SU}$</th>
<th>Affects $t_{CO}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure that the appropriate constraints are set for the failing I/Os (refer to the “Initial Compilation: Required Settings” section in the Design Optimization Overview chapter of the Quartus Prime Handbook.)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Use timing-driven compilation for I/O (Fast Input, Output, and Output Enable Registers)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Use fast input register (Programmable Delays)</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Use fast output register, fast output enable register, and fast OCT register (Programmable Delays)</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>Decrease the value of Input Delay from Pin to Input Register or set Decrease Input Delay to Input Register = ON</td>
<td>Yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Timing-Driven Compilation

This option moves registers into I/O elements if required to meet $t_{SU}$ or $t_{CO}$ assignments, duplicating the register if necessary (as in the case in which a register fans out to multiple output locations). This option is turned on by default and is a global setting. The option does not apply to MAX II series devices because they do not contain I/O registers.

The **Optimize IOC Register Placement for Timing** option affects only pins that have a $t_{SU}$ or $t_{CO}$ requirement. Using the I/O register is possible only if the register directly feeds a pin or is fed directly by a pin. This setting does not affect registers with any of the following characteristics:

- Have combinational logic between the register and the pin
- Are part of a carry or cascade chain
- Have an overriding location assignment
- Use the asynchronous load port and the value is not 1 (in device families where the port is available)

Registers with the characteristics listed are optimized using the regular Quartus Prime Fitter optimizations.

**Related Information**

- **Optimize IOC Register Placement for Timing Logic Option online help**

### Fast Input, Output, and Output Enable Registers

Normally, with correct timing assignments, the Fitter already places the I/O registers in the correct I/O cell or in the core, to meet the performance requirement. However, you can place individual registers in I/O cells manually by making fast I/O assignments with the Assignment Editor.
For more information about the Fast Input Register option, Fast Output Register option, Fast Output Enable Register option, and Fast OCT (on-chip termination) Register option, refer to Quartus Prime Help.

In MAX II series devices, which have no I/O registers, these assignments lock the register into the LAB adjacent to the I/O pin if there is a pin location assignment for that I/O pin.

If the fast I/O setting is on, the register is always placed in the I/O element. If the fast I/O setting is off, the register is never placed in the I/O element. This is true even if the Optimize IOC Register Placement for Timing option is turned on. If there is no fast I/O assignment, the Quartus Prime software determines whether to place registers in I/O elements if the Optimize IOC Register Placement for Timing option is turned on.

You can also use the four fast I/O options (Fast Input Register, Fast Output Register, Fast Output Enable Register, and Fast OCT Register) to override the location of a register that is in a LogicLock Plus region and force it into an I/O cell. If you apply this assignment to a register that feeds multiple pins, the register is duplicated and placed in all relevant I/O elements. In MAX II series devices, the register is duplicated and placed in each distinct LAB location that is next to an I/O pin with a pin location assignment.

**Programmable Delays**

You can use various programmable delay options to minimize the $t_{SU}$ and $t_{CO}$ times. For Arria, Cyclone, MAX II, MAX V, and Stratix series devices, the Quartus Prime software automatically adjusts the applicable programmable delays to help meet timing requirements. Programmable delays are advanced options to use only after you compile a project, check the I/O timing, and determine that the timing is unsatisfactory. For detailed information about the effect of these options, refer to the device family handbook or data sheet.

After you have made a programmable delay assignment and compiled the design, you can view the implemented delay values for every delay chain for every I/O pin in the Delay Chain Summary section of the Compilation Report.

You can assign programmable delay options to supported nodes with the Assignment Editor. You can also view and modify the delay chain setting for the target device with the Chip Planner and Resource Property Editor. When you use the Resource Property Editor to make changes after performing a full compilation, recompiling the entire design is not necessary; you can save changes directly to the netlist. Because these changes are made directly to the netlist, the changes are not made again automatically when you recompile the design. The change management features allow you to reapply the changes on subsequent compilations.

Although the programmable delays in newer devices are user-controllable, Altera recommends their use for advanced users only. However, the Quartus Prime software might use the programmable delays internally during the Fitter phase.

For details about the programmable delay logic options available for Altera devices, refer to the following Quartus Prime Help topics:

- Input Delay from Pin to Input Register logic option
- Input Delay from Pin to Internal Cells logic option
- Output Enable Pin Delay logic option
- Delay from Output Register to Output Pin logic option
Use PLLs to Shift Clock Edges

Using a PLL typically improves I/O timing automatically. If the timing requirements are still not met, most devices allow the PLL output to be phase shifted to change the I/O timing. Shifting the clock backwards gives a better $t_{H}$ at the expense of $t_{SU}$, while shifting it forward gives a better $t_{SU}$ at the expense of $t_{H}$. You can use this technique only in devices that offer PLLs with the phase shift option.

Figure 9-5: Shift Clock Edges Forward to Improve $t_{SU}$ at the Expense of $t_{H}$

You can achieve the same type of effect in certain devices by using the programmable delay called Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations.

Related Information
Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations Logic Option online help

Use Fast Regional Clock Networks and Regional Clocks Networks

Altera devices have a variety of hierarchical clock structures. These include dedicated global clock networks, regional clock networks, fast regional clock networks, and periphery clock networks. The available resources differ between the various Altera device families.

For the number of clocking resources available in your target device, refer to the appropriate device handbook.

In general, fast regional clocks have less delay to I/O elements than regional and global clocks, and are used for high fan-out control signals. Regional clocks provide the lowest clock delay and skew for logic contained in a single quadrant. Placing clocks on these low-skew and low-delay clock nets provides better $t_{CO}$ performance.

Spine Clock Limitations

Global clock networks, regional clock networks, and periphery clock networks have an additional level of clock hierarchy known as spine clocks. Spine clocks drive the final row and column clocks to their registers; thus, the clock to every register in the chip is reached through spine clocks. Spine clocks are not directly user controllable.

If your project has high clock routing demands, due to limitations in the Quartus Prime software, you may see spine clock errors. These errors are often seen with designs using multiple memory interfaces and high-speed serial interface (HSSI) channels (especially PMA Direct mode).

To reduce these spine clock errors, you can constrain your design to better use your regional clock resources using the following techniques:

- If your design does not use LogicLock Plus regions, or if the LogicLock Plus regions are not aligned to your clock region boundaries, create additional LogicLock Plus regions and further constrain your logic.

Note: Register packing, a Fitter optimization option, may ignore LogicLock Plus regions. If this occurs, disable register packing for specific instances through the Quartus Prime Assignment Editor.
- Some periphery features may ignore LogicLock Plus region assignments. When this happens, the global promotion process may not function properly. To ensure that the global promotion process uses the correct locations, assign specific pins to the I/Os using these periphery features.
- By default, some IP MegaCore functions apply a global signal assignment with a value of dual-regional clock. If you constrain your logic to a regional clock region and set the global signal assignment to Regional instead of Dual-Regional, you can reduce clock resource contention.

**Change How Hold Times are Optimized for MAX II Devices**
For MAX II devices, you can use the Guarantee I/O Paths Have Zero Hold Time at Fast Corner option to control how hold time is optimized by the Quartus Prime software.

**Register-to-Register Timing Optimization Techniques (LUT-Based Devices)**
The next stage of design optimization is to improve register-to-register (f\textsubscript{MAX}) timing. The following sections provide available options if the performance requirements are not achieved after compilation.

Coding style affects the performance of your design to a greater extent than other changes in settings. Always evaluate your code and make sure to use synchronous design practices.

*Note:* When using the TimeQuest analyzer, register-to-register timing optimization is the same as maximizing the slack on the clock domains in your design. You can use the techniques described in this section to improve the slack on different timing paths in your design.

Before optimizing your design, understand the structure of your design as well as the type of logic affected by each optimization. An optimization can decrease performance if the optimization does not benefit your logic structure.

**Related Information**
Recommended Design Practices documentation
Details about synchronous design practices and coding styles

**Optimize Source Code**
In many cases, optimizing the design’s source code can have a very significant effect on your design performance. In fact, optimizing your source code is typically the most effective technique for improving the quality of your results and is often a better choice than using LogicLock Plus or location assignments.

Be aware of the number of logic levels needed to implement your logic while you are coding. Too many levels of logic between registers could result in critical paths failing timing. Try restructuring the design to use pipelining or more efficient coding techniques. Also, try limiting high fan-out signals in the source code. When possible, duplicate and pipeline control signals. Make sure the duplicate registers are protected by a preserve attribute, to avoid merging during synthesis.

If the critical path in your design involves memory or DSP functions, check whether you have code blocks in your design that describe memory or functions that are not being inferred and placed in dedicated logic. You might be able to modify your source code to cause these functions to be placed into high-performance dedicated memory or resources in the target device. When using RAM/DSP blocks, enable the optional input and output registers.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus Prime software, you can check the State Machine report under Analysis & Synthesis in the Compilation Report. This report provides details, including state
encoding for each state machine that was recognized during compilation. If your state machine is not recognized, you might have to change your source code to enable it to be recognized.

Related Information

- **Recommended HDL Coding Styles documentation**
  Coding style guidelines including examples of HDL code for inferring memory, functions, guidelines, and sample HDL code for state machines
- **AN 584: Timing Closure Methodology for Advanced FPGA Designs application note.**

**Improving Register-to-Register Timing Summary**

The choice of options and settings to improve the timing margin (slack) or to improve register-to-register timing depends on the failing paths in the design. To achieve the results that best approximate your performance requirements, apply the following techniques and compile the design after each step:

1. Ensure that your timing assignments are complete and correct. For details, refer to the “Initial Compilation: Required Settings” section in the **Design Optimization Overview** chapter of the *Quartus Prime Handbook*.
2. Ensure that you have reviewed all warning messages from your initial compilation and check for ignored timing assignments.
3. Apply netlist synthesis optimization options.
4. To optimize for speed, apply the following synthesis options:
   - **Optimize Synthesis for Speed, Not Area**
   - **Flatten the Hierarchy During Synthesis**
   - **Set the Synthesis Effort to High**
   - **Change State Machine Encoding**
   - **Prevent Shift Register Inference**
   - **Use Other Synthesis Options Available in Your Synthesis Tool**
5. To optimize for performance using physical synthesis, apply the following options:
   - Perform physical synthesis for combinational logic
   - Perform automatic asynchronous signal pipelining
   - Perform register duplication
   - Perform register retiming
   - Perform logic to memory mapping
6. Try different Fitter seeds. If there are very few paths that are failing by small negative slack, then you can try with a different seed to see if there is a fit that meets constraints in the Fitter seed noise.
   
   **Note:** Omit this step if a large number of critical paths are failing or if the paths are failing badly.
7. To control placement, make LogicLock Plus assignments.
8. Make design source code modifications to fix areas of the design that are still failing timing requirements by significant amounts.
9. Make location assignments, or as a last resort, perform manual placement by back-annotating the design.

   You can use Design Space Explorer II (DSE) to automate the process of running several different compilations with different settings.

   If these techniques do not achieve performance requirements, additional design source code modifications might be required.
Related Information

Design Space Explorer II online help

Physical Synthesis Optimizations

The Quartus Prime software offers physical synthesis optimizations that can help improve the performance of many designs regardless of the synthesis tool used. Physical synthesis optimizations can be applied both during synthesis and during fitting.

Physical synthesis optimizations that occur during the synthesis stage of the Quartus Prime compilation operate either on the output from another EDA synthesis tool or as an intermediate step in Quartus Prime integrated synthesis. These optimizations make changes to the synthesis netlist to improve either area or speed, depending on your selected optimization technique and effort level.

To view and modify the synthesis netlist optimization options, click Assignments > Settings > Compiler Settings > Advanced Settings (Fitter).

If you use a third-party EDA synthesis tool and want to determine if the Quartus Prime software can remap the circuit to improve performance, you can use the Perform WYSIWYG Primitive Resynthesis option. This option directs the Quartus Prime software to unmap the LEs in an atom netlist to logic gates and then map the gates back to Altera-specific primitives. Using Altera-specific primitives enables the Fitter to remap the circuits using architecture-specific techniques.

The Quartus Prime technology mapper optimizes the design to achieve maximum speed performance, minimum area usage, or balances high performance and minimal logic usage, according to the setting of the Optimization Technique option. Set this option to Speed or Balanced.

The physical synthesis optimizations occur during the Fitter stage of the Quartus Prime compilation. Physical synthesis optimizations make placement-specific changes to the netlist that improve speed performance results for a specific Altera device.

The following physical synthesis optimizations are available during the Fitter stage for improving performance:

- Physical synthesis for combinational logic
- Automatic asynchronous signal pipelining
- Physical synthesis for registers
  - Register duplication
  - Register retiming

Note: If you want the performance gain from physical synthesis only on parts of your design, you can apply the physical synthesis options on specific instances.

To apply physical synthesis assignments for fitting on a per-instance basis, use the Quartus Prime Assignment Editor. The following assignments are available as instance assignments:

- Perform physical synthesis for combinational logic
- Perform register duplication for performance
- Perform register retiming for performance
- Perform automatic asynchronous signal pipelining

Related Information

- Perform WYSIWYG Primitive Resynthesis Logic Option online help
Turn Off Extra-Effort Power Optimization Settings

If PowerPlay power optimization settings are set to Extra Effort, your design performance can be affected. If improving timing performance is more important than reducing power use, set the PowerPlay power optimization setting to Normal.

Related Information
- PowerPlay Power Optimization Logic Option online help
- Power Optimization documentation on page 10-1

Optimize Synthesis for Speed, Not Area

The manner in which the design is synthesized has a large impact on design performance. Design performance varies depending on the way the design is coded, the synthesis tool used, and the options specified when synthesizing. Change your synthesis options if a large number of paths are failing or if specific paths are failing badly and have many levels of logic.

Set your device and timing constraints in your synthesis tool. Synthesis tools are timing-driven and optimized to meet specified timing requirements. If you do not specify a target frequency, some synthesis tools optimize for area.

Some synthesis tools offer an easy way to instruct the tool to focus on speed instead of area.

You can also specify this logic option for specific modules in your design with the Assignment Editor while leaving the default Optimization Technique setting at Balanced (for the best trade-off between area and speed for certain device families) or Area (if area is an important concern). You can also use the Speed Optimization Technique for Clock Domains option in the Assignment Editor to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

To achieve best performance with push-button compilation, follow the recommendations in the following sections for other synthesis settings. You can use DSE II to experiment with different Quartus Prime synthesis options to optimize your design for the best performance.

Related Information
- Optimization Technique Logic Option online help
- Design Space Explorer II online help

Flatten the Hierarchy During Synthesis

Synthesis tools typically let you preserve hierarchical boundaries, which can be useful for verification or other purposes. However, the best optimization results generally occur when the synthesis tool optimizes across hierarchical boundaries, because doing so often allows the synthesis tool to perform the most logic minimization, which can improve performance. Whenever possible, flatten your design hierarchy to achieve the best results.

Set the Synthesis Effort to High

Some synthesis tools offer varying synthesis effort levels to trade off compilation time with synthesis results. Set the synthesis effort to high to achieve best results when applicable.
Duplicate Logic for Fan-Out Control

Duplicating logic or registers can help improve timing in cases where moving a register in a failing timing path to reduce routing delay creates other failing paths or where there are timing problems due to the fan-out of the registers. Most often, timing failures occur not because of the high fan-out registers, but because of the location of those registers. Duplicating registers, where source and destination registers are physically close, can help improve slack on critical paths.

Many synthesis tools support options or attributes that specify the maximum fan-out of a register. When using Quartus Prime integrated synthesis, you can set the Maximum Fan-Out logic option in the Assignment Editor to control the number of destinations for a node so that the fan-out count does not exceed a specified value. You can also use the maxfan attribute in your HDL code. The software duplicates the node as required to achieve the specified maximum fan-out.

Logic duplication using Maximum Fan-Out assignments normally increases resource utilization and can potentially increase compilation time, depending on the placement and the total resource usage within the selected device. The improvement in timing performance that results because of Maximum Fan-Out assignments is very design-specific. This is because when you use the Maximum Fan-Out assignment, although the Fitter duplicates the source logic to limit the fan-out, it may not be able to control the destinations that each of the duplicated sources drive. Since the Maximum Fan-Out destination does not specify which of the destinations the duplicated source should drive, it is possible that it might still be driving logic located all around the device. To avoid this situation, you could use the Manual Logic Duplication logic option.

If you are using Maximum Fan-Out assignments, Altera recommends benchmarking your design with and without these assignments to evaluate whether they give the expected improvement in timing performance. Use the assignments only when you get improved results.

You can manually duplicate registers in the Quartus Prime software regardless of the synthesis tool used. To duplicate a register, apply the Manual Logic Duplication logic option to the register with the Assignment Editor.

Note: Various Fitter optimizations may cause a small violation to the Maximum Fan-Out assignments to improve timing.

Related Information
Manual Logic Duplication Logic Option online help

Prevent Shift Register Inference

In some cases, turning off the inference of shift registers increases performance. Doing so forces the software to use logic cells to implement the shift register instead of implementing the registers in memory blocks using the ALTSHIFT_TAPS IP core. If you implement shift registers in logic cells instead of memory, logic utilization is increased.

Use Other Synthesis Options Available in Your Synthesis Tool

With your synthesis tool, experiment with the following options if they are available:

- Turn on register balancing or retiming
- Turn on register pipelining
- Turn off resource sharing

These options can increase performance, but typically increase the resource utilization of your design.
Fitter Seed

The Fitter seed affects the initial placement configuration of the design. Changing the seed value changes the Fitter results because the fitting results change whenever there is a change in the initial conditions. Each seed value results in a somewhat different fit, and you can experiment with several different seeds to attempt to obtain better fitting results and timing performance.

When there are changes in your design, there is some random variation in performance between compilations. This variation is inherent in placement and routing algorithms—there are too many possibilities to try them all and get the absolute best result, so the initial conditions change the compilation result.

Note: Any design change that directly or indirectly affects the Fitter has the same type of random effect as changing the seed value. This includes any change in source files, Compiler Settings or Timing Analyzer Settings. The same effect can appear if you use a different computer processor type or different operating system, because different systems can change the way floating point numbers are calculated in the Fitter.

If a change in optimization settings slightly affects the register-to-register timing or number of failing paths, you cannot always be certain that your change caused the improvement or degradation, or whether it could be due to random effects in the Fitter. If your design is still changing, running a seed sweep (compiling your design with multiple seeds) determines whether the average result has improved after an optimization change and whether a setting that increases compilation time has benefits worth the increased time (such as setting the Physical Synthesis Effort to Extra). The sweep also shows the amount of random variation to expect for your design.

If your design is finalized, you can compile your design with different seeds to obtain one optimal result. However, if you subsequently make any changes to your design, you might need to perform seed sweep again.

On the Assignments menu, select Compiler Settings to control the initial placement with the seed. You can use the DSE II to perform a seed sweep easily.

You can use the following Tcl command from a script to specify a Fitter seed:

```
set_global_assignment -name SEED <value>
```

Related Information

Design Space Explorer II online help
Information about compiling your design with different seeds using Design Space Explorer II

Set Maximum Router Timing Optimization Level

To improve routability in designs where the router did not pick up the optimal routing lines, set the Router Timing Optimization Level to Maximum. This setting determines how aggressively the router tries to meet the timing requirements. Setting this option to Maximum can increase design speed slightly at the cost of increased compilation time. Setting this option to Minimum can reduce compilation time at the cost of slightly reduced design speed. The default value is Normal.

Related Information

Router Timing Optimization Level Logic Option online help
LogicLock Plus Assignments

Using LogicLock Plus assignments to improve timing performance is only recommended for older Altera devices, such as the MAX II family. For other device families, especially for larger devices such as Arria and Stratix series devices, Altera does not recommend using LogicLock Plus assignments to improve timing performance. For these devices, use the LogicLock Plus feature for performance preservation and to floorplan your design.

LogicLock Plus assignments do not always improve the performance of the design. In many cases, you cannot improve upon results from the Fitter by making location assignments. If there are existing LogicLock Plus assignments in your design, remove the assignments if your design methodology permits it. Recompile the design, and then check if the assignments are making the performance worse.

When making LogicLock Plus assignments, it is important to consider how much flexibility to give the Fitter. LogicLock Plus assignments provide more flexibility than hard location assignments. Assignments that are more flexible require higher Fitter effort, but reduce the chance of design overconstraint. The following types of LogicLock Plus assignments are available, listed in the order of decreasing flexibility:

- Auto size, floating location regions
- Fixed size, floating location regions
- Fixed size, locked location regions

If you are unsure of how big or where a LogicLock Plus region should go, the Auto/Floating options are useful for your first pass. After you determine where a LogicLock Plus region must go, modify the Fixed/Locked regions, as Auto/Floating LogicLock Plus regions can hurt your overall performance. To determine what to put into a LogicLock Plus region, refer to the timing analysis results and analyze the critical paths in the Chip Planner. The register-to-register timing paths in the Timing Analyzer section of the Compilation Report help you recognize patterns.

Related Information
- Analyzing and Optimizing the Design Floorplan with the Chip Planner on page 12-1

Hierarchy Assignments

For a design with the hierarchy shown in the figure, which has failing paths in the timing analysis results similar to those shown in the table, mod_A is probably a problem module. In this case, a good strategy to fix the failing paths is to place the mod_A hierarchy block in a LogicLock Plus region so that all the nodes are closer together in the floorplan.

Figure 9-6: Design Hierarchy
### Table 9-3: Failing Paths in a Module Listed in Timing Analysis

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_A</td>
<td>reg1</td>
</tr>
<tr>
<td>mod_A</td>
<td>reg3</td>
</tr>
<tr>
<td>mod_A</td>
<td>reg4</td>
</tr>
<tr>
<td>mod_A</td>
<td>reg7</td>
</tr>
<tr>
<td>mod_A</td>
<td>reg0</td>
</tr>
</tbody>
</table>

#### Related Information
- [Analyzing and Optimizing the Design Floorplan with the Chip Planner](#) on page 12-1

#### Location Assignments

If a small number of paths are failing to meet their timing requirements, you can use hard location assignments to optimize placement. Location assignments are less flexible for the Quartus Prime Fitter than LogicLock Plus assignments. In some cases, when you are familiar with your design, you can enter location constraints in a way that produces better results.

**Note:** Improving fitting results, especially for larger devices, such as Arria and Stratix series devices, can be difficult. Location assignments do not always improve the performance of the design. In many cases, you cannot improve upon the results from the Fitter by making location assignments.

#### Metastability Analysis and Optimization Techniques

Metastability problems can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the designer cannot guarantee that the signal will meet its setup and hold time requirements. The mean time between failures (MTBF) is an estimate of the average time between instances when metastability could cause a design failure.

You can use the Quartus Prime software to analyze the average MTBF due to metastability when a design synchronizes asynchronous signals and to optimize the design to improve the MTBF. These metastability features are supported only for designs constrained with the TimeQuest analyzer, and for select device families.

If the MTBF of your design is low, refer to the Metastability Optimization section in the Timing Optimization Advisor, which suggests various settings that can help optimize your design in terms of metastability.

This chapter describes how to enable metastability analysis and identify the register synchronization chains in your design, provides details about metastability reports, and provides additional guidelines for managing metastability.

#### Related Information
- [Understanding Metastability in FPGAs](#) white paper
- [Managing Metastability with the Quartus Prime Software](#) documentation
Periphery to Core Register Placement and Routing Optimization

The Periphery to Core Register Placement and Routing Optimization (P2C) option specifies whether the Fitter performs targeted placement and routing optimization on direct connections between periphery logic and registers in the FPGA core. P2C is an optional pre-routing-aware placement optimization stage that enables you to more reliably achieve timing closure.

**Note:** The **Periphery to Core Register Placement and Routing Optimization** option applies in both directions, periphery to core and core to periphery.

Transfers between external interfaces (for example, high-speed I/O or serial interfaces) and the FPGA often require routing many connections with tight setup and hold timing requirements. When this option is turned on, the Fitter performs P2C placement and routing decisions before those for core placement and routing. This reserves the necessary resources to ensure that your design achieves its timing requirements and avoids routing congestion for transfers with external interfaces.

This option is available as a global assignment, or can be applied to specific instances within your design.

**Figure 9-7: Periphery to Core Register Placement and Routing Optimization (P2C) Flow**

P2C runs after periphery placement, and generates placement for core registers on corresponding P2C/C2P paths, and core routing to and from these core registers.

### Related Information
- Setting Periphery to Core Optimizations in the Advanced Fitter Setting Dialog Box on page 9-23
- Setting Periphery to Core Optimizations in the Assignment Editor on page 9-24
- Viewing Periphery to Core Optimizations in the Fitter Report on page 9-25

### Setting Periphery to Core Optimizations in the Advanced Fitter Setting Dialog Box

The **Periphery to Core Placement and Routing Optimization** setting specifies whether the Fitter should perform targeted placement and routing optimization on direct connections between periphery logic and registers in the FPGA core.
You can optionally perform periphery to core optimizations by instance with settings in the Assignment Editor.

1. In the Quartus Prime software, click **Assignments > Settings > Compiler Settings > Advanced Settings (Fitter)**.

2. In the **Advanced Fitter Settings** dialog box, for the **Periphery to Core Placement and Routing Optimization** option, select one of the following options depending on how you want to direct periphery to core optimizations in your design:
   
a. Select **Auto** to direct the software to automatically identify transfers with tight timing windows, place the core registers, and route all connections to or from the periphery.
   
b. Select **On** to direct the software to globally optimize all transfers between the periphery and core registers, regardless of timing requirements.

   **Note:** Setting this option to **On** in the **Advanced Fitter Settings** is not recommended. The intended use for this setting is in the Assignment Editor to force optimization for a targeted set of nodes or instance.
   
c. Select **Off** to disable periphery to core path optimization in your design.

Related Information

- [Periphery to Core Register Placement and Routing Optimization](#) on page 9-23
- [Setting Periphery to Core Optimizations in the Assignment Editor](#) on page 9-24

### Setting Periphery to Core Optimizations in the Assignment Editor

When you turn on the **Periphery to Core Placement and Routing Optimization** (P2C/C2P) setting in the Assignment Editor, the Quartus Prime software performs periphery to core, or core to periphery optimizations on selected instances in your design.

You can optionally perform periphery to core optimizations by instance with settings in the **Advanced Fitter Settings** dialog box.

1. In the Quartus Prime software, click **Assignments > Assignment Editor**.

2. For the selected path, double-click the **Assignment Name** column, and then click the **Periphery to core register placement and routing optimization** option in the drop-down list.

3. In the **To** column, choose either a periphery node or core register node on a P2C/C2P path you want to optimize. Leave the **From** column empty.

   For paths to appear in the Assignments Editor, you must first run Analysis & Synthesis on your design.
Viewing Periphery to Core Optimizations in the Fitter Report

The Quartus Prime software generates a periphery to core placement and routing optimization summary in the Fitter (Place & Route) report after compilation.

1. Compile your Quartus Prime project.
2. In the Tasks pane, select Compilation.
3. Under Fitter (Place & Route), double-click View Report.
4. In the Fitter folder, expand the Place Stage folder.
5. Double-click Periphery to Core Transfer Optimization Summary.

Table 9-4: Fitter Report - Periphery to Core Transfer Optimization (P2C) Summary

<table>
<thead>
<tr>
<th>From Path</th>
<th>To Path</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td>Node 2</td>
<td>Placed and Routed — Core register is locked. Periphery to core/core to periphery routing is committed.</td>
</tr>
<tr>
<td>Node 3</td>
<td>Node 4</td>
<td>Placed but not Routed — Core register is locked. Routing is not committed. This occurs when P2C is not able to optimize all targeted paths within a single group, for example, the same delay/wire requirement, or the same control signals. Partial P2C routing commitments may cause unresolvable routing congestion.</td>
</tr>
</tbody>
</table>
From Path | To Path | Status
--- | --- | ---
Node 5 | Node 6 | Not Optimized—This occurs when P2C is set to Auto and the path is not optimized due to one of the following issues:

1. The delay requirement is impossible to achieve.
2. The minimum delay requirement (for hold timing) is too large. The P2C algorithm cannot efficiently handle cases when many wires need to be added to meet hold timing.
3. P2C encountered unresolvable routing congestion for this particular path.

Related Information
Periphery to Core Register Placement and Routing Optimization on page 9-23

Design Evaluation for Timing Closure

Follow the guidelines in this section when you encounter timing failures in a design. The guidelines show you how to evaluate compilation results of a design and how to address some of the problems. While the guideline does not cover specific examples of restructuring RTL to improve design speed, the analysis techniques help you to evaluate changes that may have to be made to RTL to close timing.

Review Compilation Results

Review Messages

After compiling your design, review the messages in each section of the compilation report. Most designs that fail timing start out with other problems that are reported as warning messages during compilation. Determine what causes a warning message, and whether the warning should be fixed or ignored. After reviewing the warning messages, review the informational messages. Take note of anything unexpected,
for example, unconnected ports, ignored constraints, missing files, and assumptions or optimizations that the software made.

Evaluate Physical Synthesis Results

If physical synthesis is enabled, the software can duplicate and retiming registers, and modify combinatorial logic during synthesis. After compilation, review the Optimization Results reports in the Analysis & Synthesis section. The reports list the optimizations performed by the physical synthesis optimizations, such as register duplication, retiming, and removal. These reports can be found in the Compilation Report panel.

Figure 9-8: Optimization Results Reports

When physical synthesis is enabled, compilation messages include a summary of the physical synthesis algorithms that were run, the performance improvement each algorithm achieved, and the elapsed time. The reported improvement is the sum of the largest improvement estimated to be achievable in each timing-critical clock domain. The values for the slack improvements can vary between compiles because of the random starting point of the compilation algorithms, but the values should be similar. The figure shows an example of the messages.

Figure 9-9: Compilation Messages
Evaluate Fitter Netlist Optimizations

The Fitter can also perform netlist optimizations to the design netlist. Major changes include register packing, duplicating or deleting logic cells, retiming registers, inverting signals, or modifying nodes in a general way such as moving an input from one logic cell to another. These reports can be found in the Netlist Optimizations results of the Fitter section, and they should also be reviewed.

Evaluate Optimization Results

After checking what optimizations were done and how they improved performance, evaluate the runtime it took to get the extra performance. To reduce compilation time, review the physical synthesis and netlist optimizations over a couple of compilations, and edit the RTL to reflect the changes that physical synthesis performed. If a particular set of registers consistently get retimed, edit the RTL to retime the registers the same way. If the changes are made to match what the physical synthesis algorithms did, the physical synthesis options can be turned off to save compile time while getting the same type of performance improvement.

Evaluate Resource Usage

Evaluate a variety of resources used in the design, including global and non-global signal usage, routing utilization, and clustering difficulty.

Global and Non-global Usage

If your design contains a lot of clocks, evaluate global and non-global signals. Determine whether global resources are being used effectively, and if not, consider making changes. These reports can be found in the Resource Section under Fitter in the Compilation Report panel. The figure shows an example of inefficient use of a global clock. The highlighted line has a single fan-out from a global clock. Assigning it to a Regional Clock would make the Global Clock available for another signal. You can ignore signals with an empty value in the Global Line Name column as the signal uses dedicated routing, and not a clock buffer.

Figure 9-10: Inefficient Use of a Global Clock

<table>
<thead>
<tr>
<th>Location</th>
<th>Fan-Out</th>
<th>Global Resource Used</th>
<th>Global Line Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRACTIONAL_PLL_X98_Y2_N0</td>
<td>1</td>
<td>Global Clock</td>
<td>GCLK7</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y2_N1</td>
<td>29044</td>
<td>Global Clock</td>
<td>GCLK6</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y13_N1</td>
<td>253103</td>
<td>Global Clock</td>
<td>GCLK8</td>
</tr>
<tr>
<td>FF_X185_Y66_N13</td>
<td>280349</td>
<td>Global Clock</td>
<td>GCLK4</td>
</tr>
<tr>
<td>FF_X185_Y66_N13</td>
<td>4887</td>
<td>Global Clock</td>
<td>GCLK5</td>
</tr>
<tr>
<td>FRACTIONAL_PLL_X98_Y11_N0</td>
<td>1</td>
<td>Global Clock</td>
<td>GCLK6</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y3_N1</td>
<td>1</td>
<td>Global Clock</td>
<td>GCLK5</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y11_N1</td>
<td>1691</td>
<td>Regional Clock</td>
<td>RCLK26</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y8_N1</td>
<td>302</td>
<td>Regional Clock</td>
<td>RCLK23</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y11_N1</td>
<td>141</td>
<td>Regional Clock</td>
<td>RCLK25</td>
</tr>
<tr>
<td>PLLOUTPUTCOUNTER_X98_Y10_N1</td>
<td>17</td>
<td>Regional Clock</td>
<td>RCLK22</td>
</tr>
</tbody>
</table>

The Non-Global High Fan-Out Signals report lists the highest fan-out nodes that are not routed on global signals. Reset and enable signals are at the top of the list. If there is routing congestion in the design, and there are high fan-out non-global nodes in the congested area, consider using global or regional signals to fan-out the nodes, or duplicate the high fan-out registers so that each of the duplicates can have fewer fan-outs. Use the Chip Planner to locate high fan-out nodes, to report routing congestion, and to determine whether the alternatives are viable.
Routing Usage

Review routing usage reported in the Fitter Resource Usage Summary report. The figure shows an example of the report.

Figure 9-11: Fitter Resource Usage Summary Report

The average interconnect usage reports the average amount of interconnect that is used, out of what is available on the device. The peak interconnect usage reports the largest amount of interconnect used in the most congested areas. Designs with an average value below 50% typically do not have any problems with routing. Designs with an average between 50-65% may have difficulty routing. Designs with an average over 65% typically have difficulty meeting timing unless the RTL is well designed to tolerate a highly utilized chip. Peak values at or above 90% are likely to have problems with timing closure; a 100% peak value indicates that all routing in an area of the device has been used, so there is a high possibility of degradation in timing performance. The figure shows the Report Routing Utilization report.

Figure 9-12: Report Routing Utilization Report

Wires Added for Hold

As part of the fitting process, the router can add wire between register paths to increase delay to meet hold time requirements. During the routing process, the router reports how much extra wire was used to meet hold time requirements. Excessive amounts of added wire can indicate problems with the constraint. Typically it would be caused by incorrect multicycle transfers, particularly between different rate clocks, and between different clock networks. The Fitter reports how much routing delay was added in the Estimated Delay Added for Hold Timing report. Specific register paths can be reviewed to view whether a delay was added to meet hold requirements.
An example of an incorrect constraint which can cause the router to add wire for hold requirements is when there is data transfer from 1x to 2x clocks. Assume the design intent is to allow two cycles per transfer. Data can arrive any time in the two destination clock cycles by adding a multicycle setup constraint as shown in the example:

```
set_multicycle_path -from 1x -to 2x -setup -end 2
```

The timing requirement is relaxed by one 2x clock cycle, as shown in the black line in the waveform in the figure.

**Figure 9-14: Timing Requirement Relaxed Waveform**

However, the default hold requirement, shown with the dashed blue line, may cause the router to add wire to guarantee that data is delayed by one cycle. To correct the hold requirement, add a multicycle constraint with a hold option.

```
set_multicycle_path -from 1x -to 2x -setup -end 2
set_multicycle_path -from 1x -to 2x -hold -end 1
```
The orange dashed line in the figure above represents the hold relationship, and no extra wire is required to delay the data.

The router can also add wire for hold timing requirements when data is transferred in the same clock domain, but between clock branches that use different buffering. Transferring between clock network types happens more often between the periphery and the core. The figure below shows a case where data is coming into a device, and uses a periphery clock to drive the source register, and a global clock to drive the destination register. A global clock buffer has larger insertion delay than a periphery clock buffer. The clock delay to the destination register is much larger than to the source register, hence extra delay is necessary on the data path to ensure that it meets its hold requirement.

**Figure 9-15: Clock Delay**

To identify cases where a path has different clock network types, review the path in the TimeQuest timing analyzer, and check nodes along the source and destination clock paths. Also, check the source and destination clock frequencies to see whether they are the same, or multiples, and whether there are multicycle exceptions on the paths. In some cases, cross-domain paths may also be false by intent, so make sure there are false path exceptions on those.

If you suspect that routing is added to fix real hold problems, then disable the **Optimize hold timing** option. Recompile the design and rerun timing analysis to uncover paths that fail hold time.

**Figure 9-16: Optimize Hold Timing Option**
Disabling the **Optimize hold timing** option is a debug step, and should be left enabled (default state) during normal compiles. Wire added for hold is a normal part of timing optimization during routing and is not always a problem.

### Evaluate Other Reports and Adjust Settings Accordingly

#### Difficulty Packing Design

In the Fitter Resource Section, under the **Resource Usage Summary**, review the **Difficulty Packing Design** report. The **Difficulty Packing Design** report details the effort level (low, medium, or high) of the Fitter to fit the design into the device, partition, and LogicLock Plus region. As the effort level of **Difficulty Packing Design** increases, timing closure gets harder. Going from medium to high can result in significant drop in performance or increase in compile time. Consider reducing logic to reduce packing difficulty.

#### Review Ignored Assignments

The **Compilation Report** includes details of any assignments ignored by the Fitter. Assignments typically get ignored if design names change, but assignments are not updated. Make sure any intended assignments are not being ignored.

#### Review Non-Default Settings

The reports from Synthesis and Fitter show non-default settings used in a compilation. Review the non-default settings to ensure the design benefits from the change.

#### Review Floorplan

Use the Chip Planner for reviewing placement. The Chip Planner can be used to locate hierarchical entities, and colors each located entity in the floorplan. Look for logic that seems out of place, based on where you would expect it to be. For example, logic that interfaces with I/Os should be close to the I/Os, and logic that interfaces with an IP or memory must be close to the IP or memory. The figure shows an example of a floorplan with color-coded entities. In the floorplan, the green block is spread apart. Check to see if those paths are failing timing, and if so, what connects to that module that could affect placement. The blue and aqua blocks are spread out and mixed together. Check and see if there are many connections between the two modules that may contribute to this. The pink logic at the bottom should interface with I/Os at the bottom edge.
Check fan-in and fan-out of a highlighted module by using the buttons on the task bar shown in the figure below.

**Figure 9-18: Fan-in and Fan-Out Buttons**

Look for signals that go a long way across the chip and see if they are contributing to timing failures.

Check global signal usage for signals that may affect logic placement. Logic feeding a global buffer may be pulled close to the buffer, away from related logic. High fan-out on non-global resource may pull logic together.

Check for routing congestion. Highly congested areas may cause logic to be spread out, and the design may be difficult to route.

**Evaluate Placement and Routing**
Review duration of parts of compile time in Fitter messages. If routing takes much more time than placement, then meeting timing may be more difficult than the placer predicted.

**Adjust Placement Effort**
Increasing the Placement Effort Multiplier to improve placement quality may be a good tradeoff at the cost of higher compile time, but the benefit is design dependent. The value should be adjusted after
reviewing and optimizing other settings and RTL. Try an increased value, up to 4, and reset to default if performance or compile time does not improve.

**Figure 9-19: Placement Effort Multiplier**

---

**Adjust Fitter Effort**

To increase effort, enable the **Standard Fit (highest effort)** option. The default **Auto Fit** option reduces Fitter effort when it estimates timing requirements are met.
Review Timing Constraints

Ensure that clocks are constrained with the correct frequency requirements. Using the `derive_pll_clocks` assignment keeps generated clock settings updated. TimeQuest can be useful in reviewing SDC constraints. For example, under Diagnostic in the Task panel, the Report Ignored Constraints report shows any incorrect names in the design, most commonly caused by changes in the design hierarchy. Use the Report Unconstrained Paths report to locate unconstrained paths. Add constraints as necessary so that the design can be optimized.

Review Details of Timing Paths

Show Timing Path Routing

Showing routing for a path can help uncover unusual routing delays. In the TimeQuest Tasks panel, enable the Report panel name option, and then select Report Timing. Then, turn on the Show routing option to show routing wires in the path.
The **Extra Fitter Information** tab shows a miniature floorplan with the path highlighted. The path can also be located in the Chip Planner for viewing routing congestion, and to view whether nodes in a path are placed close together or far apart.

**Global Network Buffers**

A routing path can be used to identify global network buffers that fail timing. Buffer locations are named according to the network they drive.

- CLK_CTRL_Gn—for Global driver
- CLk_CTRL_Rn—for Regional driver

Buffers to access the global networks are located in the center of each side of the device. The buffering to route a core logic signal on a global signal network will cause insertion delay. Some trade offs to consider for global and non-global routing are source location, insertion delay, fan-out, distance a signal travels, and possible congestion if the signal is demoted to local routing.

**Source Location**

If the register feeding the global buffer cannot be moved closer, then consider changing either the design logic or the routing type.

**Insertion Delay**

If a global signal is required, consider adding half a cycle to timing by using a negative-edge triggered register to generate the signal (top figure) and use a multicycle setup constraint (bottom figure).

**Figure 9-22: Negative-Edge Triggered Register**

![Negative-Edge Triggered Register](image1)

**Figure 9-23: Multicycle Setup Constraint**

![Multicycle Setup Constraint](image2)

**Fan-Out**

Nodes with very high fan-out that use local routing tend to pull logic that they drive close to the source node. This can make other paths fail timing. Duplicating registers can help reduce the impact of high fan-out paths. Consider manually duplicating and preserving these registers. Using a `MAX_FANOUT` assignment...
may make arbitrary groups of fan-out nodes, whereas a designer can make more intelligent fan-out groups.

**Global Networks**

If a signal should use a different type of global signal than it has automatically been assigned, use the Global Signal assignment to control the global signal usage on a per-signal basis. For example, if local routing is desired, set the Global Signal assignment to OFF.

**Figure 9-24: Global Signal Assignment**

<table>
<thead>
<tr>
<th>To</th>
<th>Assignment Name</th>
<th>Value</th>
<th>Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg_clk</td>
<td>Global Signal</td>
<td>Off</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Resets and Global Networks**

Reset signals are often routed on global networks. Sometimes, the use of a global network causes recovery failures. Consider reviewing the placement of the register that generates the reset and the routing path of the signal.

**Suspicious Setup**

Suspicious setup failures include paths with very small or very large requirements. One typical cause is math precision error. For example, 10Mhz/3 = 33.33 ns per period. In three cycles, the time would be 99.999 ns vs 100.000 ns. Setting a maximum delay could provide an appropriate setup relationship.

Another cause of failure would be paths that should be false by design intent, such as:

- asynchronous paths that are handled through FIFOs, or
- slow asynchronous paths that rely on handshaking for data that remain available for multiple clock cycles.

To prevent the Fitter from having to meet unnecessarily restrictive timing requirements, consider adding false or multicycle path statements.

**Logic Depth**

The Statistics tab in the TimeQuest path report shows the levels of logic in a path. If the path fails timing and the number of logic levels is high, consider adding pipelining in that part of the design.

**Auto Shift Register Replacement**

Shift registers or register chains can be converted to RAM during synthesis to save area. However, conversion to RAM often reduces speed. The names of the converted registers will include "altshift_taps".

If paths that fail timing begin or end in shift registers, consider disabling the Auto Shift Register Replacement option. Registers that are intended for pipelining should not be converted. For shift registers that are converted to a chain, evaluate area/speed trade off of implementing in RAM or logic cells. If a design is close to full, shift register conversion to RAM may benefit non-critical clock domains by saving area. The settings can be changed globally or on a register or hierarchy basis from the default of AUTO to OFF.

**Clocking Architecture**

Review the clock region boundaries in the Chip Planner. You must place registers driven by a regional clock in one quadrant of the chip.
Timing failure can occur when the I/O interface at the top of the device connects to logic driven by a regional clock which is in one quadrant of the device, and placement restrictions force long paths to and from some of the I/Os to logic across quadrants.

Use a different type of clock source to drive the logic - global, which covers the whole device, or dual-regional which covers half the device. Alternatively, you can reduce the frequency of the I/O interface to accommodate the long path delays. You can also redesign the pinout of the device to place all the specified I/Os adjacent to the regional clock quadrant. This issue can happen when register locations are restricted, such as with LogicLock Plus regions, clocking resources, or hard blocks (memories, DSPs, IPs). The Extra Fitter Information tab in the TimeQuest report informs you when placement is restricted for nodes in a path.

**Timing Closure Recommendations**

The Report Timing Closure Recommendations task in the TimeQuest analyzer analyzes paths and provides specific recommendations based on path characteristics.
Making Adjustments and Recompiling

Look for obvious problems that you can fix with minimal effort. To identify where the Compiler had trouble meeting timing, perform seed sweeping with about five compiles. Doing so shows consistently failing paths. Consider recoding or redesigning that part of the design.

To reach timing closure, a well written RTL can be more effective than changing your compilation settings. Seed sweeping can also be useful if the timing failure is very small, and the design has already been optimized for performance improvements and is close to final release. Additionally, seed sweeping can be used for evaluating changes to compilation settings. Compilation results vary due to the random nature of fitter algorithms. If a compilation setting change produces lower average performance, undo the change.

Sometimes, settings or constraints can cause more problems than they fix. When significant changes to the RTL or design architecture have been made, compile periodically with default settings and without LogicLock Plus regions, and re-evaluate paths that fail timing.

Partitioning often does not help timing closure, and should be done at the beginning of the design process. Adding partitions can increase logic utilization if it prevents cross-boundary optimizations, making timing closure harder and increasing compile times.

Adding LogicLock Plus regions can be an effective part of timing closure, but must be done at the beginning of a design. Adding new LogicLock Plus regions at the end of the design cycle can restrict placement, hence lowering the performance.

Scripting Support

You can run procedures and make settings described in this manual 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
```

You can specify many of the options described in this section either in an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <.qsf variable name> <value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <.qsf variable name> <value> -to <instance name>
```

**Note:** If the `<value>` field includes spaces (for example, ‘Standard Fit’), you must enclose the value in straight double quotation marks.

**Related Information**

Tcl Scripting documentation on page 5-1

Quartus Prime Settings Reference File Manual
Initial Compilation Settings

Use the Quartus Prime Settings File (.qsf) variable name in the Tcl assignment to make the setting along with the appropriate value. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.

The top table lists the .qsf variable name and applicable values for the settings described in the “Initial Compilation: Required Settings” section in the Design Optimization Overview chapter in the Quartus Prime Handbook. The bottom table lists the advanced compilation settings.

Table 9-5: Initial Compilation Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize IOC Register Placement For Timing</td>
<td>OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING</td>
<td>ON, OFF</td>
<td>Global</td>
</tr>
<tr>
<td>Optimize Hold Timing</td>
<td>OPTIMIZE_HOLD_TIMING</td>
<td>OFF, IO PATHS AND MINIMUM TPD PATHS, ALL PATHS</td>
<td>Global</td>
</tr>
</tbody>
</table>

Table 9-6: Advanced Compilation Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Router Timing Optimization level</td>
<td>ROUTER_TIMING_OPTIMIZATION_LEVEL</td>
<td>NORMAL, MINIMUM, MAXIMUM</td>
<td>Global</td>
</tr>
</tbody>
</table>

Resource Utilization Optimization Techniques (LUT-Based Devices)

The table lists the .qsf file variable name and applicable values for the settings described in Optimizing Timing (LUT-Based Devices).

Table 9-7: Resource Utilization Optimization Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Packed Registers (1)</td>
<td>AUTO_PACKETED_REGISTERS_&lt;device family name&gt;</td>
<td>OFF, NORMAL, MINIMIZE AREA, MINIMIZE AREA WITH CHAINS, AUTO</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Setting Name</td>
<td>.qsf File Variable Name</td>
<td>Values</td>
<td>Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>Physical Synthesis for Combinational Logic for Reducing Area</td>
<td>PHYSICAL_SYNTHESIS_COMBO_LOGIC_FOR_AREA</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Physical Synthesis for Mapping Logic to Memory</td>
<td>PHYSICAL_SYNTHESIS_MAP_LOGIC_TO_MEMORY_FOR_AREA</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Optimization Technique</td>
<td>&lt;device family name&gt;_OPTIMIZATION_TECHNIQUE</td>
<td>AREA, SPEED, BALANCED</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Speed Optimization Technique for Clock Domains</td>
<td>SYNTH_CRITICAL_CLOCK</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
<tr>
<td>State Machine Encoding</td>
<td>STATE_MACHINE_PROCESSING</td>
<td>AUTO, ONE-HOT, GRAY, JOHNSON, MINIMAL BITS, SEQUENTIAL, USER-ENCOD</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto RAM Replacement</td>
<td>AUTO_RAM_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto ROM Replacement</td>
<td>AUTO_ROM_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto Shift Register Replacement</td>
<td>AUTO_SHIFT_REGISTER_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto Block Replacement</td>
<td>AUTO_DSP_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Number of Processors for Parallel Compilation</td>
<td>NUM_PARALLEL_PROCESSORS</td>
<td>Integer between 1 and 16 inclusive, or ALL</td>
<td>Global</td>
</tr>
</tbody>
</table>

Note to table:
1. Allowed values for this setting depend on the device family that you select.

I/O Timing Optimization Techniques (LUT-Based Devices)

The table lists the .qsf file variable name and applicable values for the I/O timing optimization settings.
### Table 9-8: I/O Timing Optimization Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize IOC Register Placement For Timing</td>
<td>OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING</td>
<td>ON, OFF</td>
<td>Global</td>
</tr>
<tr>
<td>Fast Input Register</td>
<td>FAST_INPUT_REGISTER</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
<tr>
<td>Fast Output Register</td>
<td>FAST_OUTPUT_REGISTER</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
<tr>
<td>Fast Output Enable Register</td>
<td>FAST_OUTPUT_ENABLE_REGISTER</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
<tr>
<td>Fast OCT Register</td>
<td>FAST_OCT_REGISTER</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
</tbody>
</table>

### Register-to-Register Timing Optimization Techniques (LUT-Based Devices)

The table lists the .qsf file variable name and applicable values for the settings described in *Register-to-Register Timing Optimization Techniques (LUT-Based Devices)*.

### Table 9-9: Register-to-Register Timing Optimization Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform Physical Synthesis for Combinational Logic</td>
<td>PHYSICAL_SYNTHESIS_COMBO_LOGIC</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform Register Duplication</td>
<td>PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform Register Retiming</td>
<td>PHYSICAL_SYNTHESIS_REGISTER_RETIMING</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform Automatic Asynchronous Signal Pipelining</td>
<td>PHYSICAL_SYNTHESISASYNCHRONOUS_SIGNAL_PIPELINING</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Physical Synthesis Effort</td>
<td>PHYSICAL_SYNTHESIS_EFFORT</td>
<td>NORMAL, EXTRA, FAST</td>
<td>Global</td>
</tr>
<tr>
<td>Fitter Seed</td>
<td>SEED</td>
<td>&lt;integer&gt;</td>
<td>Global</td>
</tr>
<tr>
<td>Maximum Fan-Out</td>
<td>MAX_FANOUT</td>
<td>&lt;integer&gt;</td>
<td>Instance</td>
</tr>
<tr>
<td>Manual Logic Duplication</td>
<td>DUPLICATE_ATOM</td>
<td>&lt;node name&gt;</td>
<td>Instance</td>
</tr>
<tr>
<td>Optimize Power during Synthesis</td>
<td>OPTIMIZE_POWER_DURING_SYNTHESIS</td>
<td>NORMAL, OFF EXTRA_EFFORT</td>
<td>Global</td>
</tr>
</tbody>
</table>
### Document Revision History

**Table 9-10: Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Added: <em>Periphery to Core Register Placement and Routing Optimization.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>• Updated location of Fitter Settings, Analysis &amp; Synthesis Settings,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Physical Synthesis Optimizations to Compiler Settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated DSE II content.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Dita conversion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed content about obsolete devices that are no longer supported in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QII software v14.0: Arria GX, Arria II, Cyclone III, Stratix II, Stratix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>III.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replaced Megafunction content with IP core content.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>• Added Design Evaluation for Timing Closure section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Optimizing Timing (Macrocell-Based CPLDs) section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Optimize Multi-Corner Timing and Fitter Aggressive Routability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Timing Analysis with the TimeQuest Timing Analyzer to show how to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>access the Report All Summaries command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Ignored Timing Constraints to include a help link to Fitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Summary Reports with the Ignored Assignment Report information.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| May 2013         | 13.0.0  | • Renamed chapter title from Area and Timing Optimization to Timing Closure and Optimization.  
• Removed design and area/resources optimization information.  
• Added the following sections:  
  Fitter Aggressive Routability Optimization.  
  Tips for Analyzing Paths from/to the Source and Destination of Critical Path.  
  Tips for Locating Multiple Paths to the Chip Planner.  
| June 2012        | 12.0.0  | • Updated “Optimize Multi-Corner Timing” on page 13–6,  
• Minor text edits throughout the chapter. |
• Updated Table 13–6  
• Added the “Spine Clock Limitations” section  
• Removed the Change State Machine Encoding section from page 19  
• Removed Figure 13-5  
• Minor text edits throughout the chapter |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| May 2011       | 11.0.0  | • Reorganized sections in “Initial Compilation: Optional Fitter Settings” section  
                 |         | • Added new information to “Resource Utilization” section               |
|                |         | • Added new information to “Duplicate Logic for Fan-Out Control” section |
|                |         | • Added links to Help                                                   |
|                |         | • Additional edits and updates throughout chapter                       |
| December 2010  | 10.1.0  | • Added links to Help                                                   |
|                |         | • Updated device support                                                |
|                |         | • Added “Debugging Timing Failures in the TimeQuest Analyzer” section   |
|                |         | • Removed Classic Timing Analyzer references                            |
|                |         | • Other updates throughout chapter                                      |
| August 2010    | 10.0.1  | Corrected link                                                          |
| July 2010      | 10.0.0  | • Moved Compilation Time Optimization Techniques section to new         |
|                |         |  *Reducing Compilation Time* chapter                                      |
|                |         | • Removed references to Timing Closure Floorplan                        |
|                |         | • Moved Smart Compilation Setting and Early Timing Estimation sections to new *Reducing Compilation Time* chapter |
|                |         | • Added Other Optimization Resources section                             |
|                |         | • Removed outdated information                                          |
|                |         | • Changed references to DSE chapter to Help links                        |
|                |         | • Linked to Help where appropriate                                       |
|                |         | • Removed Referenced Documents section                                  |

### Related Information

**Quartus Handbook Archive**

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

The Quartus Prime software offers power-driven compilation to fully optimize device power consumption. Power-driven compilation focuses on reducing your design’s total power consumption using power-driven synthesis and power-driven place-and-route.

This chapter describes the power-driven compilation feature and flow in detail, as well as low power design techniques that can further reduce power consumption in your design. The techniques primarily target Arria®, Stratix®, and Cyclone® series of devices. These devices utilize a low-k dielectric material that dramatically reduces dynamic power and improves performance. Arria series, Stratix IV, and Stratix V device families include efficient logic structures called adaptive logic modules (ALMs) that obtain maximum performance while minimizing power consumption. Cyclone device families offer the optimal blend of high performance and low power in a low-cost FPGA.

Altera provides the Quartus Prime PowerPlay Power Analyzer to aid you during the design process by delivering fast and accurate estimations of power consumption. You can minimize power consumption, while taking advantage of the industry’s leading FPGA performance, by using the tools and techniques described in this chapter.

Total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. This chapter focuses on design optimization options and techniques that help reduce core dynamic power and I/O power. In addition to these techniques, there are additional power optimization techniques available for specific devices. These techniques include:

- Programmable Power Technology
- Device Speed Grade Selection

Related Information

- **Literature and Technical Documentation**
  For more information about a device-specific architecture, refer to the device handbook on the Altera website.

- **PowerPlay Power Analysis**
  For more information about the PowerPlay Power Analyzer, refer to volume 3 of the Quartus Prime Handbook.

- **AN 514: Power Optimization in Stratix IV FPGAs**
  For more information about power optimization techniques available for Stratix IV devices.
Power Dissipation

You can refine techniques that reduce power consumption in your design by understanding the sources of power dissipation.

The following figure shows the power dissipation of Stratix and Cyclone devices in different designs. All designs were analyzed at a fixed clock rate of 100 MHz and exhibited varied logic resource utilization across available resources.

Figure 10-1: Average Core Dynamic Power Dissipation

Notes:
1. 103 different designs were used to obtain these results.
2. 96 different designs were used to obtain these results.
3. In designs using DSP blocks, DSPs consumed 5% of core dynamic power.

In Stratix and Cyclone device families, a series of column and row interconnect wires of varying lengths provide signal interconnections between logic array blocks (LABs), memory block structures, and digital signal processing (DSP) blocks or multiplier blocks. These interconnects dissipate the largest component of device power.

FPGA combinational logic is another source of power consumption. The basic building block of logic in the latest Stratix series devices is the ALM, and in Cyclone IV GX devices, it is the logic element (LE).

For more information about ALMs and LEs in Cyclone or Stratix devices, refer to the respective device handbook.

Memory and clock resources are other major consumers of power in FPGAs. Stratix devices feature the TriMatrix memory architecture. TriMatrix memory includes 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks, which are configurable to support many features. Stratix IV TriMatrix on-chip memory is an enhancement based upon the Stratix II FPGA TriMatrix memory and includes three sizes of memory blocks: MLAB blocks, M9K blocks, and M144K blocks. Stratix IV and Stratix V devices feature Programmable Power Technology, an advanced architecture that enables a smooth trade-off between speed and power. The core of each Stratix IV and Stratix V device is divided into tiles, each of which may be put into a high-speed or low-power mode. The primary benefit of Programmable Power
Technology is to reduce static power, with a secondary benefit being a small reduction in dynamic power. Cyclone IV GX devices have 9-Kbit M9K memory blocks.

**Design Space Explorer II**

Design Space Explorer II (DSE) is an easy-to-use, self-guided design optimization utility that is included in the Quartus Prime software. DSE II explores and reports optimal Quartus Prime software options for your design, targeting either power optimization, design performance, or area utilization improvements. You can use DSE II to implement the techniques described in this chapter.

![Design Space Explorer II User Interface](image)

The power optimizations, under **Exploration mode**, target overall design power improvements. These settings focus on applying different options that specifically reduce total design thermal power.

By default, the Quartus Prime PowerPlay Power Analyzer is run for every exploration performed by DSE II when power optimizations are selected. This helps you debug your design and determine trade-offs between power requirements and performance optimization.

DSE II automatically tries different combinations of netlist optimizations and advanced Quartus Prime software compiler settings, and reports the best settings for your design, based on your chosen primary
optimization goal. You can try different seeds with DSE II if you are fairly close to meeting your timing or area requirements and find one seed that meets timing or area requirements. Finally, DSE II can run compilations on a remote compute farm, which shortens the timing closure process.

- Name your DSE II session and specify the type of compilation to perform.
- Set Exploration Points and specify Exploration mode and the number and types of Seeds to use.
- Specify the Design File Setup including the use of a specified Quartus Archive File (.qar) or create a new one.
- Specify Limits to the operation of DSE II.
- Specify the type of Results to save.
- When using a remote compute farm, DSE II uses the values in the DSE Server Settings box to specify a registration host and network ports to connect.
- Options in the Advanced settings allow you to specify options such as:
  - Turn on the option to specify exploration points without compiling.
  - Specify the Maximum number of parallel compilations used by DSE II.
  - Specify the Maximum number of CPUs that can be used by DSE II.
  - Specify a quality of fit formula.

When you have completed your configuration, you can perform an exploration by clicking Start.

Related Information
Launch the Design Space Explorer II
For more information about the DSE II, refer to Quartus Prime Help.

Power-Driven Compilation


Quartus Prime software settings that control power-driven compilation are located in the PowerPlay power optimization during synthesis list in the Advanced Settings (Synthesis) dialog box, and the PowerPlay power optimization during fitting list on the Advanced Fitter Settings dialog box. The following sections describes these power optimization options at the Analysis and Synthesis and Fitter levels.

Power-Driven Synthesis

Synthesis netlist optimization occurs during the synthesis stage of the compilation flow. The optimization technique makes changes to the synthesis netlist to optimize your design according to the selection of area, speed, or power optimization. This section describes power optimization techniques at the synthesis level.

To access the PowerPlay Power Optimization During Synthesis option, click Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis).

You can apply these settings on a project or entity level.
Table 10-1: Optimize Power During Synthesis Options

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>No netlist, placement, or routing optimizations are performed to minimize power.</td>
</tr>
<tr>
<td>Normal compilation (Default)</td>
<td>Low compute effort algorithms are applied to minimize power through netlist optimizations as long as they are not expected to reduce design performance.</td>
</tr>
<tr>
<td>Extra effort</td>
<td>High compute effort algorithms are applied to minimize power through netlist optimizations. Max performance might be impacted.</td>
</tr>
</tbody>
</table>

The **Normal compilation** setting is turned on by default. This setting performs memory optimization and power-aware logic mapping during synthesis.

Memory blocks can represent a large fraction of total design dynamic power. Minimizing the number of memory blocks accessed during each clock cycle can significantly reduce memory power. Memory optimization involves effective movement of user-defined read/write enable signals to associated read- and write-clock enable signals for all memory types.

A default implementation of a simple dual-port memory block in which write-clock enable signals and read-clock enable signals are connected to $V_{CC}$, making both read and write memory ports active during each clock cycle.

**Figure 10-3: Memory Transformation**

Memory transformation effectively moves the read-enable and write-enable signals to the respective read-clock enable and write-clock enable signals. By using this technique, memory ports are shut down when they are not accessed. This significantly reduces your design’s memory power consumption. For Stratix IV and Stratix V devices, the memory transformation takes place at the Fitter level by selecting the **Normal compilation** settings for the power optimization option.

In Cyclone IV GX and Stratix IV devices, the specified read-during-write behavior can significantly impact the power of single-port and bidirectional dual-port RAMs. It is best to set the read-during-write parameter to “Don’t care” (at the HDL level), as it allows an optimization whereby the read-enable signal can be set to the inversion of the existing write-enable signal (if one exists). This allows the core of the RAM to shut down (that is, not toggle), which saves a significant amount of power.

The other type of power optimization that takes place with the **Normal compilation** setting is power-aware logic mapping. The power-aware logic mapping reduces power by rearranging the logic during synthesis to eliminate nets with high toggle rates.
The **Extra effort** setting performs the functions of the **Normal compilation** setting and other memory optimizations to further reduce memory power by shutting down memory blocks that are not accessed. This level of memory optimization can require extra logic, which can reduce design performance.

The **Extra effort** setting also performs power-aware memory balancing. Power-aware memory balancing automatically chooses the best memory configuration for your memory implementation and provides optimal power saving by determining the number of memory blocks, decoder, and multiplexer circuits required. If you have not previously specified target-embedded memory blocks for your design’s memory functions, the power-aware balancer automatically selects them during memory implementation.

The following figure is an example of a 4k × 4 (4k deep and 4 bits wide) memory implementation in two different configurations using M4K memory blocks available in some Stratix devices.

**Figure 10-4: 4K × 4 Memory Implementation Using Multiple M4K Blocks**

![Memory Implementation Diagram]

The minimum logic area implementation uses M4K blocks configured as 4k × 1. This implementation is the default in the Quartus Prime software because it has the minimum logic area (0 logic cells) and the highest speed. However, all four M4K blocks are active on each memory access in this implementation, which increases RAM power. The minimum RAM power implementation is created by selecting **Extra effort** in the **PowerPlay power optimization** list. This implementation automatically uses four M4K blocks configured as 1k × 4 for optimal power saving. An address decoder is implemented by the RAM megafunction to select which of the four M4K blocks should be activated on a given cycle, based on the state of the top two user address bits. The RAM megafunction automatically implements a multiplexer to feed the downstream logic by choosing the appropriate M4K output. This implementation reduces RAM power because only one M4K block is active on any cycle, but it requires extra logic cells, costing logic area and potentially impacting design performance.

There is a trade-off between power saved by accessing fewer memories and power consumed by the extra decoder and multiplexor logic. The Quartus Prime software automatically balances the power savings against the costs to choose the lowest power configuration for each logical RAM. The benchmark data shows that the power-driven synthesis can reduce memory power consumption by as much as 60% in Stratix devices.

Memory optimization options can also be controlled by the **Low_Power_Mode** parameter in the **Default Parameters** page of the **Settings** dialog box. The settings for this parameter are **None**, **Auto**, and **ALL**. **None** corresponds to the **Off** setting in the **PowerPlay power optimization** list. **Auto** corresponds to the **Normal compilation** setting and **ALL** corresponds to the **Extra effort** setting, respectively. You can apply PowerPlay power optimization either on a compiler basis or on individual entities.
parameter always takes precedence over the **Optimize Power for Synthesis** option for power optimization on memory.

You can also set the `MAXIMUM_DEPTH` parameter manually to configure the memory for low power optimization. This technique is the same as the power-aware memory balancer, but it is manual rather than automatic like the **Extra effort** setting in the **PowerPlay power optimization** list. You can set the `MAXIMUM_DEPTH` parameter for memory modules manually in the megafunction instantiation or in the IP Catalog for power optimization. The `MAXIMUM_DEPTH` parameter always takes precedence over the **Optimize Power for Synthesis** options for power optimization on memory optimization.

Related Information
**Reducing Memory Power Consumption** on page 10-11
For more information about clock enable signals.

**Power-Driven Fitter**

The **Compiler Settings** page provides access to **PowerPlay power optimization** settings.

You can apply these settings only on a project-wide basis. The **Extra effort** setting for the Fitter requires extensive effort to optimize the design for power and can increase the compilation time.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>No netlist, placement, or routing optimizations are performed to minimize power.</td>
</tr>
<tr>
<td>Normal compilation</td>
<td>Low compute effort algorithms are applied to minimize power through placement and routing optimizations as long as they are not expected to reduce design performance.</td>
</tr>
<tr>
<td>(Default)</td>
<td></td>
</tr>
<tr>
<td>Extra effort</td>
<td>High compute effort algorithms are applied to minimize power through placement and routing optimizations. Max performance might be impacted.</td>
</tr>
</tbody>
</table>

The **Normal compilation** setting is selected by default and performs DSP optimization by creating power-efficient DSP block configurations for your DSP functions. For Stratix IV and Stratix V devices, this setting, which is based on timing constraints entered for the design, enables the Programmable Power Technology to configure tiles as high-speed mode or low-power mode. Programmable Power Technology is always turned **ON** even when the **OFF** setting is selected for the **PowerPlay power optimization** option. Tiles are the combination of LAB and MLAB pairs (including the adjacent routing associated with LAB and MLAB), which can be configured to operate in high-speed or low-power mode. This level of power optimization does not have any affect on the fitting, timing results, or compile time.

The **Extra effort** setting performs the functions of the **Normal compilation** setting and other place-and-route optimizations during fitting to fully optimize the design for power. The Fitter applies an extra effort to minimize power even after timing requirements have been met by effectively moving the logic closer during placement to localize high-toggling nets, and using routes with low capacitance. However, this effort can increase the compilation time.

The **Extra effort** setting uses a Value Change Dump File (**.vcd**) that guides the Fitter to fully optimize the design for power, based on the signal activity of the design. The best power optimization during fitting results from using the most accurate signal activity information. Signal activities from full post-fit netlist (timing) simulation provide the highest accuracy because all node activities reflect the actual design behavior, provided that supplied input vectors are representative of typical design operation. If you do not have a **.vcd** file, the Quartus Prime software uses assignments, clock assignments, and vectorless estimation values (PowerPlay Power Analyzer Tool settings) to estimate the signal activities. This information is used to optimize your design for power during fitting. The benchmark data shows that the
power-driven Fitter technique can reduce power consumption by as much as 19% in Stratix devices. On average, you can reduce core dynamic power by 16% with the Extra effort synthesis and Extra effort fitting settings, as compared to the Off settings in both synthesis and Fitter options for power-driven compilation.

**Note:** Only the Extra effort setting in the PowerPlay power optimization list for the Fitter option uses the signal activities (from .vcd files) during fitting. The settings made in the PowerPlay Power Analyzer Settings page in the Settings dialog box are used to calculate the signal activity of your design.

**Related Information**
- **AN 514: Power Optimization in Stratix IV FPGAs**
  For more information about Stratix IV power optimization.
- **PowerPlay Power Analysis**
  For more information about .vcd files and how to create them, refer to the Quartus Prime Handbook.

### Area-Driven Synthesis

Using area optimization rather than timing or delay optimization during synthesis saves power because you use fewer logic blocks. Using less logic usually means less switching activity. The Quartus Prime integrated synthesis tool provides Speed, Balanced, or Area for the Optimization Technique option. You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the Area setting (potentially at the expense of register-to-register timing performance) while leaving the default Optimization Technique setting at Balanced (for the best trade-off between area and speed for certain device families). The Speed Optimization Technique can increase the resource usage of your design if the constraints are too aggressive, and can also result in increased power consumption.

The benchmark data shows that the area-driven technique can reduce power consumption by as much as 31% in Stratix devices and as much as 15% in Cyclone devices.

### Gate-Level Register Retiming

You can also use gate-level register retiming to reduce circuit switching activity. Retiming shuffles registers across combinational blocks without changing design functionality. The Perform gate-level register retiming option in the Quartus Prime software enables the movement of registers across combinational logic to balance timing, allowing the software to trade off the delay between timing critical and noncritical timing paths.

Retiming uses fewer registers than pipelining. In this example of gate-level register retiming, the 10 ns critical delay is reduced by moving the register relative to the combinational logic, resulting in the reduction of data depth and switching activity.
Figure 10-5: Gate-Level Register Retiming

![Gate-Level Register Retiming Diagram]

**Note:** Gate-level register retiming makes changes at the gate level. If you are using an atom netlist from a third-party synthesis tool, you must also select the **Perform WYSIWYG primitive resynthesis** option to undo the atom primitives to gates mapping (so that register retiming can be performed), and then to remap gates to Altera primitives. When using Quartus Prime integrated synthesis, retiming occurs during synthesis before the design is mapped to Altera primitives. The benchmark data shows that the combination of WYSIWYG remapping and gate-level register retiming techniques can reduce power consumption by as much as 6% in Stratix devices and as much as 21% in Cyclone devices.

**Related Information**

- [Netlist Optimizations and Physical Synthesis](#) on page 13-1
  For more information about register retiming, refer to the *Quartus Prime Handbook*.

**Design Guidelines**

Several low-power design techniques can reduce power consumption when applied during FPGA design implementation. This section provides detailed design techniques for Cyclone IV GX devices that affect overall design power. The results of these techniques might be different from design to design.

**Clock Power Management**

Clocks represent a significant portion of dynamic power consumption due to their high switching activity and long paths. Actual clock-related power consumption is higher than this because the power consumed by local clock distribution within logic, memory, and DSP or multiplier blocks is included in the power consumption for the respective blocks.

Clock routing power is automatically optimized by the Quartus Prime software, which enables only those portions of the clock network that are required to feed downstream registers. Power can be further reduced by gating clocks when they are not required. It is possible to build clock-gating logic, but this approach is not recommended because it is difficult to generate a glitch free clock in FPGAs using ALMs or LEs.

Cyclone IV, Stratix IV, and Stratix V devices use clock control blocks that include an enable signal. A clock control block is a clock buffer that lets you dynamically enable or disable the clock network and dynamically switch between multiple sources to drive the clock network. You can use the Quartus Prime IP Catalog to create this clock control block with the ALTCLKCTRL megafunction. Cyclone IV,
Stratix IV, and Stratix V devices provide clock control blocks for global clock networks. In addition, Stratix IV, and Stratix V devices have clock control blocks for regional clock networks. The dynamic clock enable feature lets internal logic control the clock network. When a clock network is powered down, all the logic fed by that clock network does not toggle, thereby reducing the overall power consumption of the device. For example, the following shows a 4-input clock control block diagram.

**Figure 10-6: Clock Control Block Diagram**

![Clock Control Block Diagram](image)

The enable signal is applied to the clock signal before being distributed to global routing. Therefore, the enable signal can either have a significant timing slack (at least as large as the global routing delay) or it can reduce the $f_{\text{MAX}}$ of the clock signal.

Another contributor to clock power consumption is the LAB clock that distributes a clock to the registers within a LAB. LAB clock power can be the dominant contributor to overall clock power. For example, in Cyclone devices, each LAB can use two clocks and two clock enable signals, as shown in the following figure. Each LAB’s clock signal and clock enable signal are linked. For example, an LE in a particular LAB using the `labclk1` signal also uses the `labclkena1` signal.

**Figure 10-7: LAB-Wide Control Signals**

![LAB-Wide Control Signals](image)

To reduce LAB-wide clock power consumption without disabling the entire clock tree, use the LAB-wide clock enable to gate the LAB-wide clock. The Quartus Prime software automatically promotes register-level clock enable signals to the LAB-level. All registers within an LAB that share a common clock and clock enable are controlled by a shared gated clock. To take advantage of these clock enables, use a clock enable construct in the relevant HDL code for the registered logic.

**Related Information**

Clock Control Block Megafunction User Guide (ALTCLKCTRL)

For more information about using clock control blocks.
LAB-Wide Clock Enable Example
This VHDL code makes use of a LAB-wide clock enable. This clock-gating logic is automatically turned into an LAB-level clock enable signal.

```vhdl
IF clk'event AND clock = '1' THEN
    IF logic_is_enabled = '1' THEN
        reg <= value;
    ELSE
        reg <= reg;
    END IF;
ELSE
    reg <= reg;
END IF;
```

Reducing Memory Power Consumption
The memory blocks in FPGA devices can represent a large fraction of typical core dynamic power. Memory consumes approximately 20% of the core dynamic power in typical some device designs. Memory blocks are unlike most other blocks in the device because most of their power is tied to the clock rate, and is insensitive to the toggle rate on the data and address lines.

When a memory block is clocked, there is a sequence of timed events that occur within the block to execute a read or write. The circuitry controlled by the clock consumes the same amount of power regardless of whether or not the address or data has changed from one cycle to the next. Thus, the toggle rate of input data and the address bus have no impact on memory power consumption.

The key to reducing memory power consumption is to reduce the number of memory clocking events. You can achieve this through clock network-wide gating, or on a per-memory basis through use of the clock enable signals on the memory ports.

The logical view of the internal clock of the memory block. Use the appropriate enable signals on the memory to make use of the clock enable signal instead of gating the clock.

Figure 10-8: Memory Clock Enable Signal

Using the clock enable signal enables the memory only when necessary and shuts it down for the rest of the time, reducing the overall memory power consumption. You can create these enable signals by selecting the Clock enable signal option for the appropriate port when generating the memory block function.
For example, consider a design that contains a 32-bit-wide M4K memory block in ROM mode that is running at 200 MHz. Assuming that the output of this block is only required approximately every four cycles, this memory block will consume 8.45 mW of dynamic power according to the demands of the downstream logic. By adding a small amount of control logic to generate a read clock enable signal for the memory block only on the relevant cycles, the power can be cut 75% to 2.15 mW.

You can also use the MAXIMUM_DEPTH parameter in your memory megafunction to save power in Cyclone IV GX, Stratix IV, and Stratix V devices; however, this approach might increase the number of LEs required to implement the memory and affect design performance.

You can set the MAXIMUM_DEPTH parameter for memory modules manually in the megafunction instantiation. The Quartus Prime software automatically chooses the best design memory configuration for optimal power.
Memory Power Reduction Example

Power usage measurements for a 4K × 36 simple dual-port memory implemented using multiple M4K blocks in a Stratix device. For each implementation, the M4K blocks are configured with a different memory depth.

Table 10-3: 4K × 36 Simple Dual-Port Memory Implemented Using Multiple M4K Blocks

<table>
<thead>
<tr>
<th>M4K Configuration</th>
<th>Number of M4K Blocks</th>
<th>ALUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K × 1 (Default setting)</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>2K × 2</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>1K × 4</td>
<td>36</td>
<td>62</td>
</tr>
<tr>
<td>512 × 9</td>
<td>32</td>
<td>143</td>
</tr>
<tr>
<td>256 × 18</td>
<td>32</td>
<td>302</td>
</tr>
<tr>
<td>128 × 36</td>
<td>32</td>
<td>633</td>
</tr>
</tbody>
</table>

Using the MAXIMUM_DEPTH parameter can save power. For all implementations, a user-provided read enable signal is present to indicate when read data is required. Using this power-saving technique can reduce power consumption by as much as 60%.
As the memory depth becomes more shallow, memory dynamic power decreases because unaddressed M4K blocks can be shut off using a decoded combination of address bits and the read enable signal. For a 128-deep memory block, power used by the extra LEs starts to outweigh the power gain achieved by using a more shallow memory block depth. The power consumption of the memory blocks and associated LEs depends on the memory configuration.

**Note:** The SOPC Builder and Qsys system do not offer specific power savings control for on-chip memory block. There is no read enable, write enable, or clock enable that you can enable in the on-chip RAM megafunction to shut down the RAM block in the SOPC Builder and Qsys system.

### Pipelining and Retiming

Designs with many glitches consume more power because of faster switching activity. Glitches cause unnecessary and unpredictable temporary logic switches at the output of combinational logic. A glitch usually occurs when there is a mismatch in input signal timing leading to unequal propagation delay.

For example, consider an input change on one input of a 2-input XOR gate from 1 to 0, followed a few moments later by an input change from 0 to 1 on the other input. For a moment, both inputs become 1 (high) during the state transition, resulting in 0 (low) at the output of the XOR gate. Subsequently, when the second input transition takes place, the XOR gate output becomes 1 (high). During signal transition, a glitch is produced before the output becomes stable.

![XOR Gate Showing Glitch at the Output](image)

This glitch can propagate to subsequent logic and create unnecessary switching activity, increasing power consumption. Circuits with many XOR functions, such as arithmetic circuits or cyclic redundancy check (CRC) circuits, tend to have many glitches if there are several levels of combinational logic between registers.

Pipelining can reduce design glitches by inserting flipflops into long combinational paths. Flipflops do not allow glitches to propagate through combinational paths. Therefore, a pipelined circuit tends to have less glitching. Pipelining has the additional benefit of generally allowing higher clock speed operations, although it does increase the latency of a circuit (in terms of the number of clock cycles to a first result).

An example where pipelining is applied to break up a long combinational path.
Pipelining is very effective for glitch-prone arithmetic systems because it reduces switching activity, resulting in reduced power dissipation in combinational logic. Additionally, pipelining allows higher-speed operation by reducing logic-level numbers between registers. The disadvantage of this technique is that if there are not many glitches in your design, pipelining can increase power consumption by adding unnecessary registers. Pipelining can also increase resource utilization. The benchmark data shows that pipelining can reduce dynamic power consumption by as much as 30% in Cyclone and Stratix devices.

**Architectural Optimization**

You can use design-level architectural optimization by taking advantage of specific device architecture features. These features include dedicated memory and DSP or multiplier blocks available in FPGA devices to perform memory or arithmetic-related functions. You can use these blocks in place of LUTs to reduce power consumption. For example, you can build large shift registers from RAM-based FIFO buffers instead of building the shift registers from the LE registers.

The Stratix device family allows you to efficiently target small, medium, and large memories with the TriMatrix memory architecture. Each TriMatrix memory block is optimized for a specific function. M512 memory blocks are more power-efficient than the distributed memory structures in some competing FPGAs. The M4K memory blocks are used to implement buffers for a wide variety of applications, including processor code storage, large look-up table implementation, and large memory applications. The M-RAM blocks are useful in applications where a large volume of data must be stored on-chip. Effective utilization of these memory blocks can have a significant impact on power reduction in your design.

The latest Stratix and Cyclone device families have configurable M9K memory blocks that provide various memory functions such as RAM, FIFO buffers, and ROM.

**Related Information**

- [Area and Timing Optimization](#) on page 9-1
  
  For more information about using DSP and memory blocks efficiently, refer to the *Quartus Prime Handbook*. 
I/O Power Guidelines

Nonterminated I/O standards such as LVTTL and LVCMOS have a rail-to-rail output swing. The voltage difference between logic-high and logic-low signals at the output pin is equal to the $V_{CCIO}$ supply voltage. If the capacitive loading at the output pin is known, the dynamic power consumed in the I/O buffer can be calculated.

$$P = 0.5 \times F \times C \times V^2$$

In this equation, $F$ is the output transition frequency and $C$ is the total load capacitance being switched. $V$ is equal to $V_{CCIO}$ supply voltage. Because of the quadratic dependence on $V_{CCIO}$, lower voltage standards consume significantly less dynamic power.

Transistor-to-transistor logic (TTL) I/O buffers consume very little static power. As a result, the total power consumed by a LVTTL or LVCMOS output is highly dependent on load and switching frequency.

When using resistively terminated I/O standards like SSTL and HSTL, the output load voltage swings by a small amount around some bias point. The same dynamic power equation is used, where $V$ is the actual load voltage swing. Because this is much smaller than $V_{CCIO}$, dynamic power is lower than for nonterminated I/O under similar conditions. These resistively terminated I/O standards dissipate significant static (frequency-independent) power, because the I/O buffer is constantly driving current into the resistive termination network. However, the lower dynamic power of these I/O standards means they often have lower total power than LVCMOS or LVTTL for high-frequency applications. Use the lowest drive strength I/O setting that meets your speed and waveform requirements to minimize I/O power when using resistively terminated standards.

You can save a small amount of static power by connecting unused I/O banks to the lowest possible $V_{CCIO}$ voltage of 1.2 V.

For more information about I/O standards, refer to the Stratix IV Device Handbook, or the Cyclone IV GX Handbook on the Altera website.

When calculating I/O power, the PowerPlay Power Analyzer uses the default capacitive load set for the I/O standard in the Capacitive Loading page of the Device and Pin Options dialog box. Any other components defined in the board trace model are not taken into account for the power measurement.

For Cyclone IV GX, Stratix IV, and Stratix V, devices, Advanced I/O Timing is always used, which uses the full board trace model.

Related Information

- **I/O Management** on page 2-1

  For information about using Advanced I/O Timing and configuring a board trace model, refer to the Quartus Prime Handbook.

Dynamically Controlled On-Chip Terminations

Stratix IV and Stratix V FPGAs offer dynamic on-chip termination (OCT). Dynamic OCT enables series termination (RS) and parallel termination (RT) to dynamically turn on/off during the data transfer. This feature is especially useful when Stratix IV and Stratix V FPGAs are used with external memory interfaces, such as interfacing with DDR memories.

Compared to conventional termination, dynamic OCT reduces power consumption significantly as it eliminates the constant DC power consumed by parallel termination when transmitting data. Parallel termination is extremely useful for applications that interface with external memories where I/O standards, such as HSTL and SSTL, are used. Parallel termination supports dynamic OCT, which is useful for bidirectional interfaces.
The following is an example of power saving for a DDR3 interface using on-chip parallel termination.

The static current consumed by parallel OCT is equal to the $V_{CCIO}$ voltage divided by 100 W. For DDR3 interfaces that use SSTL-15, the static current is $1.5 \text{ V} \times 15 \text{ mA} = 22.5 \text{ mW}$. For an interface with 72 DQ and 18 DQS pins, the static power is $90 \text{ pins} \times 22.5 \text{ mW} = 2.025 \text{ W}$. Dynamic parallel OCT disables parallel termination during write operations, so if writing occurs 50% of the time, the power saved by dynamic parallel OCT is $50\% \times 2.025 \text{ W} = 1.0125 \text{ W}$.

**Related Information**

**Stratix IV Device I/O Features**

For more information about dynamic OCT in Stratix IV devices, refer to the chapter in the *Stratix IV Device Handbook*.

**Power Optimization Advisor**

The Quartus Prime software includes the Power Optimization Advisor, which provides specific power optimization advice and recommendations based on the current design project settings and assignments. The advisor covers many of the suggestions listed in this chapter. The following example shows how to reduce your design power with the Power Optimization Advisor.

**Power Optimization Advisor Example**

After compiling your design, run the PowerPlay Power Analyzer to determine your design power and to see where power is dissipated in your design. Based on this information, you can run the Power Optimization Advisor to implement recommendations that can reduce design power.

The Power Optimization Advisor after compiling a design that is not fully optimized for power.

**Figure 10-14: Power Optimization Advisor**

The Power Optimization Advisor shows the recommendations that can reduce power in your design. The recommendations are split into stages to show the order in which you should apply the recommended settings. The first stage shows mostly CAD setting options that are easy to implement and highly effective in reducing design power. An icon indicates whether each recommended setting is made in the current project. The checkmark icons for Stage 1 shows the recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icon shows the general suggestions. Each recommendation includes the description, summary of the effect of the recommendation, and the action required to make the appropriate setting.
There is a link from each recommendation to the appropriate location in the Quartus Prime user interface where you can change the setting. After making the recommended changes, recompile your design. The Power Optimization Advisor indicates with green check marks that the recommendations were implemented successfully. You can use the PowerPlay Power Analyzer to verify your design power results.

Figure 10-15: Implementation of Power Optimization Advisor Recommendations

The recommendations listed in Stage 2 generally involve design changes, rather than CAD settings changes as in Stage 1. You can use these recommendations to further reduce your design power consumption. Altera recommends that you implement Stage 1 recommendations first, then the Stage 2 recommendations.

Document Revision History

Table 10-4: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated screenshot for DSE II GUI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information about remote hosts for DSE II.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td></td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>• Updated location of Fitter Settings, Analysis &amp; Synthesis Settings,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Physical Synthesis Optimizations to Compiler Settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated DSE II GUI and optimization settings.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>Updated the format.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>Added a note to “Memory Power Reduction Example” on Qsys and SOPC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Builder power savings limitation for on-chip memory block.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Was chapter 11 in the 9.1.0 release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figures 14-2, 14-3, 14-6, 14-18, 14-19, and 14-20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated device support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Updated Figure 11-1 and associated references</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated device support</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial update</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Was chapter 9 in the 8.1.0 release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated for the Quartus Prime software release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added benchmark results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed several sections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 13–1, Figure 13–17, and Figure 13–18</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>• Changed to 8½” × 11” page size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed references to altsyncram to RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Added support for Stratix IV devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Table 9–1 and 9–9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Architectural Optimization” on page 9–22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added “Dynamically-Controlled On-Chip Terminations” on page 9–26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Referenced Documents” on page 9–29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated references</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
This chapter describes techniques to reduce resource usage when designing for Altera devices.

Resource Utilization

Determining device utilization is important regardless of whether your design achieved a successful fit. If your compilation results in a no-fit error, resource utilization information is important for analyzing the fitting problems in your design. If your fitting is successful, review the resource utilization information to determine whether the future addition of extra logic or other design changes might introduce fitting difficulties. Also, review the resource utilization information to determine if it is impacting timing performance.

To determine resource usage, refer to the Flow Summary section of the Compilation Report. This section reports resource utilization, including pins, memory bits, digital signal processing (DSP) blocks, and phase-locked loops (PLLs). Flow Summary indicates whether your design exceeds the available device resources. More detailed information is available by viewing the reports under Resource Section in the Fitter section of the Compilation Report.

Flow Summary shows the overall logic utilization. The Fitter can spread logic throughout the device, which may lead to higher overall utilization.

As the device fills up, the Fitter automatically searches for logic functions with common inputs to place in one ALM. The number of packed registers also increases. Therefore, a design that has high overall utilization might still have space for extra logic if the logic and registers can be packed together more tightly.

The reports under the Resource Section in the Fitter section of the Compilation Report provide more detailed resource information. The Fitter Resource Usage Summary report breaks down the logic utilization information and provides other resource information, including the number of bits in each type of memory block. This panel also contains a summary of the usage of global clocks, PLLs, DSP blocks, and other device-specific resources.

You can also view reports describing some of the optimizations that occurred during compilation. For example, if you use Quartus Prime integrated synthesis, the reports in the Optimization Results folder in the Analysis & Synthesis section include information about registers that integrated synthesis removed during synthesis. Use this report to estimate device resource utilization for a partial design to ensure that registers were not removed due to missing connections with other parts of the design.
If a specific resource usage is reported as less than 100% and a successful fit cannot be achieved, either there are not enough routing resources or some assignments are illegal. In either case, a message appears in the **Processing** tab of the Messages window describing the problem.

If the Fitter finishes unsuccessfully and runs much faster than on similar designs, a resource might be over-utilized or there might be an illegal assignment. If the Quartus Prime software seems to run for an excessively long time compared to runs on similar designs, a legal placement or route probably cannot be found. In the Compilation Report, look for errors and warnings that indicate these types of problems.

You can use the Chip Planner to find areas of the device that have routing congestion on specific types of routing resources. If you find areas with very high congestion, analyze the cause of the congestion. Issues such as high fan-out nets not using global resources, an improperly chosen optimization goal (speed versus area), very restrictive floorplan assignments, or the coding style can cause routing congestion. After you identify the cause, modify the source or settings to reduce routing congestion.

**Related Information**
- **Fitter Resources Report**
  For more information about Fitter Resources Report
- **Analyzing and Optimizing the Design Floorplan with the Chip Planner** on page 12-1
  For details about using the Chip Planner tool

### Optimizing Resource Utilization (LUT-Based Devices)

The following lists the stages after design analysis:

- Optimize resource utilization—Ensure that you have already set the basic constraints
- I/O timing optimization—Optimize I/O timing after you optimize resource utilization and your design fits in the desired target device
- Register-to-register timing optimization

**Related Information**

- **Design Optimization Overview** on page 7-1
  Provides information about setting basic constraints
- **Timing Closure and Optimization** on page 9-1
  Provides information about optimizing I/O timing. These tips are valid for all FPGA families and the MAX II family of CPLDs.

### Using the Resource Optimization Advisor

The Resource Optimization Advisor provides guidance in determining settings that optimize resource usage. To run the Resource Optimization Advisor, on the Tools menu, point to **Advisors**, and click **Resource Optimization Advisor**.

The Resource Optimization Advisor provides step-by-step advice about how to optimize resource usage (logic element, memory block, DSP block, I/O, and routing) of your design. Some of the recommenda-
tions in these categories might conflict with each other. Altera recommends evaluating the options and choosing the settings that best suit your requirements.

Related Information
Resource Optimization Advisor Command Tools Menu
For more information about the Resource Optimization Advisor

Resolving Resource Utilization Issues Summary
Resource utilization issues can be divided into the following three categories:

Table 11-1: Resource Utilization Issues

<table>
<thead>
<tr>
<th>Types of Resource Utilization Issues</th>
<th>Refer to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issues relating to I/O pin utilization or placement, including dedicated I/O blocks such as PLLs or LVDS transceivers</td>
<td>I/O Pin Utilization or Placement on page 11-3</td>
</tr>
<tr>
<td>Issues relating to logic utilization or placement, including logic cells containing registers and LUTs as well as dedicated logic, such as memory blocks and DSP blocks</td>
<td>Logic Utilization or Placement on page 11-4</td>
</tr>
<tr>
<td>Issues relating to routing</td>
<td>Routing on page 11-8</td>
</tr>
</tbody>
</table>

Related Information
- I/O Pin Utilization or Placement on page 11-3
- Logic Utilization or Placement on page 11-4
- Routing on page 11-8

I/O Pin Utilization or Placement
Resolve I/O resource problems with these guidelines.

Guideline: Use I/O Assignment Analysis
To help with pin placement, on the Processing menu, point to Start and click Start I/O Assignment Analysis. The Start I/O Assignment Analysis command allows you to check your I/O assignments early in the design process. You can use this command to check the legality of pin assignments before, during, or after compilation of your design. If design files are available, you can use this command to accomplish more thorough legality checks on your design’s I/O pins and surrounding logic. These checks include proper reference voltage pin usage, valid pin location assignments, and acceptable mixed I/O standards.

Common issues with I/O placement relate to the fact that differential standards have specific pin pairings and certain I/O standards might be supported only on certain I/O banks.

If your compilation or I/O assignment analysis results in specific errors relating to I/O pins, follow the recommendations in the error message. Right-click the message in the Messages window and click Help to open the Quartus Prime Help topic for this message.
Guideline: Modify Pin Assignments or Choose a Larger Package

If a design that has pin assignments fails to fit, compile the design without the pin assignments to determine whether a fit is possible for the design in the specified device and package. You can use this approach if a Quartus Prime error message indicates fitting problems due to pin assignments.

If the design fits when all pin assignments are ignored or when several pin assignments are ignored or moved, you might have to modify the pin assignments for the design or select a larger package.

If the design fails to fit because insufficient I/Os pins are available, a successful fit can often be obtained by using a larger device package (which can be the same device density) that has more available user I/O pins.

Related Information
- I/O Management on page 2-1
  For more information about I/O assignment analysis

Logic Utilization or Placement

Resolve logic resource problems, including logic cells containing registers and LUTs, as well as dedicated logic such as memory blocks and DSP blocks, with these guidelines.

Guideline: Optimize Source Code

If your design does not fit because of logic utilization, then evaluate and modify the design at the source. You can often improve logic significantly by making design-specific changes to your source code. This is typically the most effective technique for improving the quality of your results.

If your design does not fit into available logic elements (LEs) or ALMs, but you have unused memory or DSP blocks, check if you have code blocks in your design that describe memory or DSP functions that are not being inferred and placed in dedicated logic. You might be able to modify your source code to allow these functions to be placed into dedicated memory or DSP resources in the target device.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus Prime software, you can check for the State Machine report under Analysis & Synthesis in the Compilation Report. This report provides details, including the state encoding for each state machine that was recognized during compilation. If your state machine is not being recognized, you might have to change your source code to enable it to be recognized.

Related Information
- Recommended HDL Coding Styles
  For coding style guidelines, including examples of HDL code for inferring memory and DSP functions and sample HDL code for state machines
- AN 584: Timing Closure Methodology for Advanced FPGA Designs.
  For additional HDL coding examples

Guideline: Optimize Synthesis for Area, Not Speed

If your design fails to fit because it uses too much logic, resynthesize the design to improve the area utilization. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Particularly when area utilization of the design is a concern, ensure that you do not over-constrain the timing requirements for the design. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.
If resource utilization is an important concern, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus Prime integrated synthesis, select **Balanced** or **Area** for the **Optimization Technique**. You can also specify an **Optimization Technique** logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing performance) while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. You can also use the **Speed Optimization Technique for Clock Domains** logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

In some synthesis tools, not specifying an \( f_{\text{MAX}} \) requirement can result in less resource utilization.

**Note:** In the Quartus Prime software, the **Balanced** setting typically produces utilization results that are very similar to those produced by the **Area** setting, with better performance results. The **Area** setting can give better results in some cases.

The Quartus Prime software provides additional attributes and options that can help improve the quality of your synthesis results.

**Related Information**

**Synthesis**
For information about setting the timing requirements and synthesis options in Quartus Prime integrated synthesis and other synthesis tools

**Guideline: Restructure Multiplexers**
Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexed logic, you can achieve a more efficient implementation in your Altera device.

**Related Information**

- **Restructure Multiplexers logic option**
  For more information about the Restructure Multiplexers option
- **Recommended HDL Coding Styles**
  For design guidelines to achieve optimal resource utilization for multiplexer designs

**Guideline: Perform WYSIWYG Primitive Resynthesis with Balanced or Area Setting**
The **Perform WYSIWYG Primitive Resynthesis** logic option specifies whether to perform WYSIWYG primitive resynthesis during synthesis. This option uses the setting specified in the **Optimization Technique** logic option. The **Perform WYSIWYG Primitive Resynthesis** logic option is useful for resynthesizing some or all of the WYSIWYG primitives in your design for better area or performance. However, WYSIWYG primitive resynthesis can be done only when you use third-party synthesis tools.

**Note:** The **Balanced** setting typically produces utilization results that are very similar to the **Area** setting with better performance results. The **Area** setting can give better results in some cases. Performing WYSIWYG resynthesis for area in this way typically reduces register-to-register timing performance.

**Related Information**

**Perform WYSIWYG Primitive Resynthesis logic option**
For information about this logic option
Guideline: Use Register Packing

The Auto Packed Registers option implements the functions of two cells into one logic cell by combining the register of one cell in which only the register is used with the LUT of another cell in which only the LUT is used.

Related Information

Auto Packed Registers logic option

For more information about the Auto Packed Registers logic option

Guideline: Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet may not fit in the targeted device. For example, a design might fail to fit if the location or LogicLock Plus assignments are too strict and not enough routing resources are available on the device.

To resolve routing congestion caused by restrictive location constraints or LogicLock Plus region assignments, use the Routing Congestion task in the Chip Planner to locate routing problems in the floorplan, then remove any internal location or LogicLock Plus region assignments in that area. If your design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock Plus assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor or the Chip Planner. To remove LogicLock Plus assignments in the Chip Planner, in the LogicLock Plus Regions Window, or on the Assignments menu, click Remove Assignments. Turn on the assignment categories you want to remove from the design in the Available assignment categories list.

Related Information

• Analyzing and Optimizing the Design Floorplan with the Chip Planner on page 12-1

   For more information about the Routing Congestion task in the Chip Planner

Guideline: Flatten the Hierarchy During Synthesis

Synthesis tools typically provide the option of preserving hierarchical boundaries, which can be useful for verification or other purposes. However, the Quartus Prime software optimizes across hierarchical boundaries so as to perform the most logic minimization, which can reduce area in a design with no design partitions.

Guideline: Retarget Memory Blocks

If your design fails to fit because it runs out of device memory resources, your design may require a certain type of memory that the device does not have. For example, a design that requires two M-RAM blocks cannot be targeted to a device with only one M-RAM block. You might be able to obtain a fit by building one of the memories with a different size memory block, such as an M4K memory block.

If the memory block was created with a parameter editor, open the parameter editor and edit the RAM block type so it targets a new memory block size.

ROM and RAM memory blocks can also be inferred from your HDL code, and your synthesis software can place large shift registers into memory blocks by inferring the Shift register (RAM-based) IP core. This inference can be turned off in your synthesis tool to cause the memory or shift registers to be placed in logic instead of in memory blocks. Also, for improved timing performance, you can turn this inference off to prevent registers from being moved into RAM.

Depending on your synthesis tool, you can also set the RAM block type for inferred memory blocks. In Quartus Prime integrated synthesis, set the ramstyle attribute to the desired memory type for the inferred
RAM blocks, or set the option to **logic**, to implement the memory block in standard logic instead of a memory block.

Consider the Resource Utilization by Entity report in the report file and determine whether there is an unusually high register count in any of the modules. Some coding styles can prevent the Quartus Prime software from inferring RAM blocks from the source code because of the blocks’ architectural implementation, and force the software to implement the logic in flipflops. As an example, a function such as an asynchronous reset on a register bank might make the resistor bank incompatible with the RAM blocks in the device architecture, so that the register bank is implemented in flipflops. It is often possible to move a large register bank into RAM by slight modification of associated logic.

**Guideline: Use Physical Synthesis Options to Reduce Area**

The physical synthesis options for fitting help you decrease resource usage. When you enable these options, the Quartus Prime software makes placement-specific changes to the netlist that reduce resource utilization for a specific Altera device.

**Note:** The compilation time might increase considerably when you use physical synthesis options.

With the Quartus Prime software, you can apply physical synthesis options to specific instances, which can reduce the impact on compilation time. Physical synthesis instance assignments allow you to enable physical synthesis algorithms for specific portions of your design.

The following physical synthesis optimizations for fitting are available:

- Physical synthesis for combinational logic
- Map logic into memory

**Related Information**

Physical Synthesis Optimizations Page Settings Dialog Box

**Guideline: Retarget or Balance DSP Blocks**

A design might not fit because it requires too many DSP blocks. You can implement all DSP block functions with logic cells, so you can retarget some of the DSP blocks to logic to obtain a fit.

If the DSP function was created with the parameter editor, open the parameter editor and edit the function so it targets logic cells instead of DSP blocks. The Quartus Prime software uses the DEDICATED_MULTIPLIER_CIRCUITRY IP core parameter to control the implementation.

DSP blocks also can be inferred from your HDL code for multipliers, multiply-adders, and multiply-accumulators. You can turn off this inference in your synthesis tool. When you are using Quartus Prime integrated synthesis, you can disable inference by turning off the **Auto DSP Block Replacement** logic option for your entire project. Click **Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis)**. Turn off **Auto DSP Block Replacement**. Alternatively, you can disable the option for a specific block with the Assignment Editor.

The Quartus Prime software also offers the **DSP Block Balancing** logic option, which implements DSP block elements in logic cells or in different DSP block modes. The default **Auto** setting allows DSP block balancing to convert the DSP block slices automatically as appropriate to minimize the area and maximize the speed of the design. You can use other settings for a specific node or entity, or on a project-wide basis, to control how the Quartus Prime software converts DSP functions into logic cells and DSP blocks. Using any value other than **Auto** or **Off** overrides the DEDICATED_MULTIPLIER_CIRCUITRY parameter used in IP core variations.

**Guideline: Use a Larger Device**

If a successful fit cannot be achieved because of a shortage of routing resources, you might require a larger device.
Routing

Resolve routing resource problems with these guidelines.

Guideline: Set Auto Packed Registers to Sparse or Sparse Auto
The Auto Packed Registers option reduces LE or ALM count in a design. You can set this option by clicking Assignment > Settings > Compiler Settings > Advanced Settings (Fitter).

Related Information
Auto Packed Registers logic option

Guideline: Set Fitter Aggressive Routability Optimizations to Always
The Fitter Aggressive Routability Optimization option is useful if your design does not fit due to excessive routing wire utilization.

If there is a significant imbalance between placement and routing time (during the first fitting attempt), it might be because of high wire utilization. Turning on the Fitter Aggressive Routability Optimizations option can reduce your compilation time.

On average, this option can save up to 6% wire utilization, but can also reduce performance by up to 4%, depending on the device.

Related Information
Fitter Aggressive Routability Optimizations logic option

Guideline: Increase Router Effort Multiplier
The Router Effort Multiplier controls how quickly the router tries to find a valid solution. The default value is 1.0 and legal values must be greater than 0. Numbers higher than 1 help designs that are difficult to route by increasing the routing effort. Numbers closer to 0 (for example, 0.1) can reduce router runtime, but usually reduce routing quality slightly. Experimental evidence shows that a multiplier of 3.0 reduces overall wire usage by approximately 2%. Using a Router Effort Multiplier higher than the default value could be beneficial for designs with complex datapaths with more than five levels of logic. However, congestion in a design is primarily due to placement, and increasing the Router Effort Multiplier does not necessarily reduce congestion.

Note: Any Router Effort Multiplier value greater than 4 only increases by 10% for every additional 1. For example, a value of 10 is actually 4.6.

Guideline: Remove Fitter Constraints
A design with conflicting constraints or constraints that are difficult to achieve may not fit the targeted device. Conflicting or difficult-to-achieve constraints can occur when location or LogicLock Plus assignments are too strict and there are not enough routing resources.

In this case, use the Routing Congestion task in the Chip Planner to locate routing problems in the floorplan, then remove all location and LogicLock Plus region assignments from that area. If the local constraints are removed, and the design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock Plus assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor or the Chip Planner. To remove LogicLock Plus assignments in the Chip Planner, in the LogicLock Plus Regions Window, or on the Assignments menu, click Remove Assignments. Turn on the assignment categories you want to remove from the design in the Available assignment categories list.
Related Information

- Analyzing and Optimizing the Design Floorplan with the Chip Planner on page 12-1
  For more information about the Routing Congestion task in the Chip Planner

Guideline: Optimize Synthesis for Area, Not Speed

In some cases, resynthesizing the design to improve the area utilization can also improve the routability of the design. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Ensure that you do not overconstrain the timing requirements for the design, particularly when the area utilization of the design is a concern. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.

If resource utilization is important to improve the routing results in your design, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus Prime integrated synthesis, click Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis). For Optimization Technique, select Balanced or Area.

You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the Area setting (potentially at the expense of register-to-register timing performance). You can apply the setting to specific modules while leaving the default Optimization Technique setting at Balanced (for the best trade-off between area and speed for certain device families) or Speed. You can also use the Speed Optimization Technique for Clock Domains logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

Note: In the Quartus Prime software, the Balanced setting typically produces utilization results that are very similar to those obtained with the Area setting, with better performance results. The Area setting can yield better results in some unusual cases.

In some synthesis tools, not specifying an $f_{\text{MAX}}$ requirement can result in less resource utilization, which can improve routability.

Related Information

Synthesis
For information about setting the timing requirements and synthesis options in Quartus Prime integrated synthesis and other synthesis tools

Guideline: Optimize Source Code

If your design does not fit because of routing problems and the methods described in the preceding sections do not sufficiently improve the routability of the design, modify the design at the source to achieve the desired results. You can often improve results significantly by making design-specific changes to your source code, such as duplicating logic or changing the connections between blocks that require significant routing resources.

Guideline: Use a Larger Device

If a successful fit cannot be achieved because of a shortage of routing resources, you might require a larger device.

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
```

You can specify many of the options described in this section either in an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <.qsf variable name> <value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <.qsf variable name> <value> -to <instance name>
```

**Note:** If the `<value>` field includes spaces (for example, ‘Standard Fit’), you must enclose the value in straight double quotation marks.

**Related Information**

Tcl Scripting on page 5-1
For more information about Tcl scripting

Quartus Prime Settings File Manual
For more information about all settings and constraints in the Quartus Prime software

Command-Line Scripting on page 4-1
For more information about command-line scripting

**Initial Compilation Settings**

Use the Quartus Prime Settings File (.qsf) variable name in the Tcl assignment to make the setting along with the appropriate value. The *Type* column indicates whether the setting is supported as a global setting, an instance setting, or both.

**Table 11-2: Advanced Compilation Settings**

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement Effort Multiplier</td>
<td>PLACEMENT_EFFORT_MULTIPLIER</td>
<td>Any positive, non-zero value</td>
<td>Global</td>
</tr>
<tr>
<td>Router Effort Multiplier</td>
<td>ROUTER_EFFORT_MULTIPLIER</td>
<td>Any positive, non-zero value</td>
<td>Global</td>
</tr>
<tr>
<td>Router Timing Optimization level</td>
<td>ROUTER_TIMING_OPTIMIZATION_LEVEL</td>
<td>NORMAL, MINIMUM, MAXIMUM</td>
<td>Global</td>
</tr>
<tr>
<td>Final Placement Optimization</td>
<td>FINAL_PLACEMENT_OPTIMIZATION</td>
<td>ALWAYS, AUTOMATICALLY, NEVER</td>
<td>Global</td>
</tr>
</tbody>
</table>
Table 11-3: Resource Utilization Optimization Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Packed Registers (1)</td>
<td>AUTO_PACKED_REGISTERS_&lt;device family name&gt;</td>
<td>OFF, NORMAL, MINIMIZE AREA, MINIMIZE AREA WITH CHAINS, AUTO</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Physical Synthesis for Combinational Logic for Reducing Area</td>
<td>PHYSICAL_SYNTHESIS_COMBO_LOGIC_FOR_AREA</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Physical Synthesis for Mapping Logic to Memory</td>
<td>PHYSICAL_SYNTHESIS_MAP_LOGIC_TO_MEMORY_FOR_AREA</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Optimization Technique</td>
<td>&lt;device family name&gt;_OPTIMIZATION_TECHNIQUE</td>
<td>AREA, SPEED, BALANCED</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Speed Optimization Technique for Clock Domains</td>
<td>SYNTH_CRITICAL_CLOCK</td>
<td>ON, OFF</td>
<td>Instance</td>
</tr>
<tr>
<td>State Machine Encoding</td>
<td>STATE_MACHINE_PROCESSING</td>
<td>AUTO, ONE-HOT, GRAY, JOHNSON, MINIMAL BITS, ONE-HOT, SEQUENTIAL, USER-ENCODE</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto RAM Replacement</td>
<td>AUTO_RAM_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto ROM Replacement</td>
<td>AUTO_ROM_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Auto Shift Register Replacement</td>
<td>AUTO_SHIFT_REGISTER_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
</tbody>
</table>

(1) Allowed values for this setting depend on the device family that you select.
<table>
<thead>
<tr>
<th>Setting Name</th>
<th>.qsf File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Block Replacement</td>
<td>AUTO_DSP_RECOGNITION</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Number of Processors for Parallel Compilation</td>
<td>NUM_PARALLEL_PROCESSORS</td>
<td>Integer between 1 and 16 inclusive, or ALL</td>
<td>Global</td>
</tr>
</tbody>
</table>

**Document Revision History**

Table 11-4: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
</tbody>
</table>
| June 2014  | 14.0.0  | • Removed Cyclone III and Stratix III devices references.  
|           |         | • Removed Macrocell-Based CPLDs related information.  
|           |         | • Updated template.                                                   |
| May 2013   | 13.0.0  | Initial release.                                                       |
Analyzing and Optimizing the Design Floorplan with the Chip Planner

As FPGA designs grow larger in density, the ability to analyze the design for performance, routing congestion, and logic placement is critical to meet the design requirements. This chapter discusses how to analyze the design floorplan with the Chip Planner.

Design floorplan analysis helps to close timing and ensure optimal performance in highly complex designs. With analysis capability, the Quartus Prime Chip Planner helps you close timing quickly on your designs. Use the Chip Planner together with LogicLock Plus regions to compile your designs hierarchically, preserving the timing results from individual compilation runs. Use LogicLock Plus regions to improve your productivity.

You can perform design analysis, as well as create and optimize the design floorplan with the Chip Planner. To make I/O assignments, use the Pin Planner.

Related Information

I/O Management on page 2-1
For information about the Pin Planner.

Altera Training
For training courses on the Chip Planner.

Migrating Assignments between Quartus Prime Standard Edition and Quartus Prime Pro Edition

The Quartus Prime Pro Edition software does not support the Quartus Prime Standard Edition LogicLock assignments and vice versa. Therefore, if you are migrating a design from Quartus Prime Pro Edition to Quartus Prime Standard Edition or vice versa, you must convert the LogicLock or LogicLock Plus assignments.

Related Information

Migrating to the Quartus Prime Pro Edition
Chip Planner Overview

The Chip Planner provides a visual display of chip resources. The Chip Planner shows:

- Logic placement regions
- LogicLock Plus regions
- Relative resource usage
- Detailed routing information
- Fan-in and fan-out connections between nodes
- Timing paths between registers
- Delay estimates for paths
- Routing congestion information

The Chip Planner lets you:

- Make assignment changes with the Chip Planner, such as creating and deleting resource assignments.
- Perform post-compilation changes such as creating, moving, and deleting logic cells and I/O atoms.
- View and create assignments for a design floorplan.
- Perform power and design analyses.
- Change connections between resources and make post-compilation changes to the properties of logic cells, I/O elements, PLLs, and RAM and digital signal processing (DSP) blocks.

Starting the Chip Planner

To start the Chip Planner, choose Tools > Chip Planner. You can also start the Chip Planner by the following methods:

- Click the Chip Planner icon on the Quartus Prime software toolbar.
- On the Shortcut menu in the following tools, click Locate > Locate in Chip Planner to locate:
  - Compilation Report
  - LogicLock Plus Regions window
  - Technology Map Viewer
  - Project Navigator window
  - RTL source code
  - Node Finder
  - Simulation Report
  - RTL Viewer
  - Report Timing panel of the TimeQuest Timing Analyzer

Chip Planner Toolbar

The Chip Planner provides powerful tools for design analysis with a GUI. You can access Chip Planner commands from the View menu and the Shortcut menu, or by clicking the icons on the toolbar.

Chip Planner Presets, Layers, and Editing Modes

The Chip Planner models types of resource objects as unique display layers. It also uses presets—which are predefined sets of layer settings—to control the display of resources.

The Chip Planner provides a set of default presets, and you can create custom presets to customize the display for your particular needs. The Basic, Detailed, and Floorplan Editing presets provided with the Chip Planner are useful for general assignment-related activities.
The Chip Planner editing modes determine the operations that you can perform. The assignment editing mode allows you to make assignment changes that are applied by the Fitter during the next place and route operation.

**Locate History Window**

As you optimize your design floorplan, you might have to locate a path or node in the Chip Planner many times. The Locate History window lists all the nodes and paths you have displayed using a Locate in Chip Planner command, providing easy access to the nodes and paths of interest to you.

If you locate a required path from the TimeQuest Timing Analyzer Report Timing pane, the Locate History window displays the required clock path. If you locate an arrival path from the TimeQuest Timing Analyzer Report Timing pane, the Locate History window displays the path from the arrival clock to the arrival data. Double-clicking a node or path in the Locate History window displays the selected node or path in the Chip Planner.

**Related Information**

Layers Settings Dialog Box

**LogicLock Plus Regions**

LogicLock Plus regions are floorplan location constraints that help you constrain logic in the target device. When you assign entity instances or nodes to a LogicLock Plus region, you direct the Fitter to place those entity instances or nodes within the region during fitting. Your floorplan can contain multiple LogicLock Plus regions.

The Quartus Prime Pro Edition software implements fully hierarchical LogicLock Plus region assignments. A LogicLock Plus region is defined by its height, width, and location. Regions can be rectangular, non-rectangular, or disjoint. You can assign two region types:

- **Placement regions**—Use these regions to constrain logic to a specific area of the device; the Fitter places the logic in the region you specify. The Fitter may also place other logic in the region unless you have designated the region as reserved.
- **Routing regions**—If you are using partial reconfiguration, use these regions to constrain routing to a specific area. You use a routing region with a corresponding placement region; the routing region must fully contain the placement region. Additionally, routing regions cannot overlap.

LogicLock Plus regions support the following additional assignments:

- **Reserved**—Prevents the Fitter from placing other logic in the region. You cannot apply the Reserved assignment to routing regions.
- **Core-Only**—Excludes pins from a region. Unlike the Quartus Prime Standard Edition software, Quartus Prime Pro Edition region assignments apply to pins by default. If the region is designated as reserved and core only, pins are not reserved from the region.

**Note:** LogicLock Plus assignments are not compatible with Quartus Prime Standard Edition LogicLock assignments. Additionally, Quartus Prime Pro Edition cannot use LogicLock assignments and Quartus Prime Standard Edition cannot use LogicLock Plus assignments.

**Creating LogicLock Plus Regions**

You can create LogicLock Plus Regions using several methods, such as with the Chip Planner.
Creating LogicLock Plus Regions with the Project Navigator

After you perform either a full compilation or analysis and elaboration on the design, the Project Navigator displays the hierarchy of the design. If the Project Navigator is not already open, choose View > Utility Windows > Project Navigator. With the design hierarchy fully expanded, right-click on any design entity, and click Create New LogicLock Plus Region to create a LogicLock Plus region and assign the entity to the new region. The new region has the same name as the entity.

Creating LogicLock Plus Regions with the LogicLock Plus Regions Window

To create a LogicLock Plus region with the LogicLock Plus Regions window, choose Assignments > LogicLock Plus Regions Window. In the LogicLock Plus Regions window, click <<new>>.

Creating LogicLock Plus Regions with the Chip Planner

To create a LogicLock Plus region in the Chip Planner, choose View > LogicLock Plus Regions > Assign LogicLock Plus Region, then click and drag on the Chip Planner floorplan to create a region of your preferred location and size.

Creating LogicLock Plus Regions with the Assignment Editor

Unlike Quartus Prime Standard Edition, in Quartus Prime Pro Edition you can assign LogicLock Plus regions using the Assignment Editor. To create an assignment, choose Spectra-Q <placement type> in the Assignment Name field.

Creating Non-Rectangular LogicLock Plus Regions

When you create a floorplan for your design, you may want to create non-rectangular LogicLock Plus regions to exclude certain resources from the LogicLock Plus region. You might also create a non-rectangular LogicLock Plus region to place certain parts of your design around specific device resources to improve performance.

To create a non-rectangular region with the Merge LogicLock Plus Region command, follow these steps:

1. In the Chip Planner, create two or more contiguous or non-contiguous rectangular regions. Do not assign logic to the regions; leave them unassigned.
2. Arrange the regions that you have created into the locations where you want the non-rectangular region.
3. Select all the individual regions that you want to merge by clicking each of them while pressing the Shift key.
4. Right-click the title bar of any of the LogicLock Plus regions that you want to merge, point to LogicLock Plus Regions, and then click Merge LogicLock Plus Region. The individual regions that you select merge to create a single new region.

You cannot merge assigned regions. First create and merge the regions, then assign logic to the merged region.

Related Information

Creating LogicLock Plus Regions on page 12-3
Hierarchical Regions

LogicLock Plus regions are fully hierarchical. Parent regions should completely contain child regions and must overlap with child regions. If the parent and child regions do not overlap, the software issues an error. The Reserved and Core-Only assignments also apply hierarchically.

LogicLock Plus assignments follow the same precedence as other constraints and assignments.

Placing Device Resources into LogicLock Plus Regions

A LogicLock Plus region includes all device resources within its boundaries, including memory and pins. The Quartus Prime Pro Edition software automatically includes pins when you assign in instance to a region. You can manually exclude pins with a core only assignment.

When you add an entity to a region, region has the same name as the entity. You cannot change the region name.

Only one LogicLock Plus region can use a device resource. If a LogicLock Plus region boundary includes part of a device resource, the Quartus Prime software allocates the entire resource to that LogicLock Plus region.

Note: If you want to import multiple instances of a module into a top-level design, you must ensure that the device has two or more locations with exactly the same device resources. (You can determine this from the applicable device specifications.) If the device does not have another area with exactly the same resources, the Quartus Prime software generates a fitting error during compilation of the top-level design.

LogicLock Plus Regions Window

Use the LogicLock Plus Regions window to create LogicLock Plus regions, assign nodes and entities to them, and modify the properties of a LogicLock Plus region. You can modify the size, state, width, height, origin, and whether the region is a reserved or core only region.

Reserved LogicLock Plus Region

The Quartus Prime software honors all entity and node assignments to LogicLock Plus regions. Occasionally, entities and nodes do not occupy an entire region, which leaves some of the region’s resources unoccupied.

To increase the region’s resource utilization and performance, the Quartus Prime software’s default behavior fills the unoccupied resources with other nodes and entities that have not been assigned to another region. You can prevent this behavior by turning on Reserved in the LogicLock Plus Regions window. When you turn on this option, your LogicLock Plus region contains only the entities and nodes that you specifically assigned to your LogicLock Plus region.

Additional Quartus Prime LogicLock Plus Design Features

To complement the LogicLock Plus Regions window, the Quartus Prime software has additional features to help you design with LogicLock Plus regions.

Analysis and Synthesis Resource Utilization by Entity

The Compilation Report contains an Analysis and Synthesis Resource Utilization by Entity section, which reports resource usage statistics, including entity-level information. You can use this feature to verify that any LogicLock Plus region you manually create contains enough resources to accommodate all the entities you assign to it.
Quartus Prime Revisions Feature

When you evaluate different LogicLock Plus regions in your design, you might want to experiment with different configurations to achieve your desired results. The Quartus Prime Revisions feature allows you to organize the same project with different settings until you find an optimum configuration.

To use the Revisions feature, choose Project > Revisions. You can create a revision from the current design or any previously created revisions. Each revision can have an associated description. You can use revisions to organize the placement constraints created for your LogicLock Plus regions.

Virtual Pins

A virtual pin is an I/O element that is temporarily mapped to a logic element and not to a pin during compilation. The software implements it as a LUT.

When you apply the Virtual Pin assignment to an input pin, the pin no longer appears as an FPGA pin, but is fixed to GND or VCC in the design. The assigned pin is not an open node.

Virtual pins should be used only for I/O elements in lower-level design entities that become nodes when imported to the top-level design. You can create virtual pins by assigning the Virtual Pin logic option to an I/O element.

You might use virtual pin assignments when you compile a partial design, because not all the I/Os from a partial design drive chip pins at the top level.

The virtual pin assignment identifies the I/O ports of a design module that are internal nodes in the top-level design. These assignments prevent the number of I/O ports in the lower-level modules from exceeding the total number of available device pins. Every I/O port that you designate as a virtual pin becomes mapped to either a logic cell or an adaptive logic module (ALM), depending on the target device.

Note: The Virtual Pin logic option must be assigned to an input or output pin. If you assign this option to a bidirectional pin, tri-state pin, or registered I/O element, Analysis & Synthesis ignores the assignment. If you assign this option to a tri-state pin, the Fitter inserts an I/O buffer to account for the tri-state logic; therefore, the pin cannot be a virtual pin. You can use multiplexer logic instead of a tri-state pin if you want to continue to use the assigned pin as a virtual pin. Do not use tri-state logic except for signals that connect directly to device I/O pins.

In the top-level design, you connect these virtual pins to an internal node of another module. By making assignments to virtual pins, you can place those pins in the same location or region on the device as that of the corresponding internal nodes in the top-level module. You can use the Virtual Pin option when compiling a LogicLock Plus module with more pins than the target device allows. The Virtual Pin option can enable timing analysis of a design module that more closely matches the performance of the module after you integrate it into the top-level design.

Note: In the Node Finder, you can set Filter Type to Pins: Virtual to display all assigned virtual pins in the design. Alternatively, to access the Node Finder from the Assignment Editor, double-click the To field; when the arrow appears on the right side of the field, click the arrow and select Node Finder.

Design Floorplan Analysis in the Chip Planner

The Chip Planner helps you visually analyze the floorplan of your design at any stage of your design cycle. With the Chip Planner, you can view post-compilation placement, connections, and routing paths.

You can also create LogicLock Plus regions and location assignments. The Chip Planner allows you to create new logic cells and I/O atoms and to move existing logic cells and I/O atoms in your design. You
can also see global and regional clock regions within the device, and the connections between I/O atoms, PLLs and the different clock regions.

From the Chip Planner, you can launch the Resource Property Editor that changes the properties and parameters of device resources and modifies connectivity between certain types of device resources. The Change Manager records any changes that you make to your design floorplan so that you can selectively undo changes.

The following sections present Chip Planner floorplan views and design analysis procedures which you can use with any Chip Planner preset, unless a procedure requires a specific preset or editing mode.

Related Information

- About the Change Manager
- About the Resource Property Editor
  For more information about the Resource Property Editor and the Change Manager, refer to Quartus Prime Help.

Chip Planner Floorplan Views

The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you zoom in, the level of abstraction decreases, revealing more details about your design.

Bird’s Eye View

The Bird’s Eye View displays a high-level picture of resource usage for the entire chip and provides a fast and efficient way to navigate between areas of interest in the Chip Planner.

The Bird’s Eye View is particularly useful when the parts of your design that you want to view are at opposite ends of the chip and you want to quickly navigate between resource elements without losing your frame of reference.

Properties Window

The Properties Window displays detailed properties of the objects (such as atoms, paths, LogicLock Plus regions, or routing elements) currently selected in the Chip Planner. To display the Properties Window, click Properties on the View menu in the Chip Planner.

Related Information

- Displaying Resources and Information
- Bird’s Eye View
  For more information about displaying information in the Bird’s Eye View.

Viewing Architecture-Specific Design Information

By adjusting the Layers Settings in the Chip Planner, you can view the following architecture-specific information related to your design:
Device routing resources used by your design—View how blocks are connected, as well as the signal routing that connects the blocks.

**LE configuration**—View logic element (LE) configuration in your design. For example, you can view which LE inputs are used; if the LE utilizes the register, the look-up table (LUT), or both; as well as the signal flow through the LE.

**ALM configuration**—View ALM configuration in your design. For example, you can view which ALM inputs are used, if the ALM utilizes the registers, the upper LUT, the lower LUT, or all of them. You can also view the signal flow through the ALM.

**I/O configuration**—View device I/O resource usage. For example, you can view which components of the I/O resources are used, if the delay chain settings are enabled, which I/O standards are set, and the signal flow through the I/O.

**PLL configuration**—View phase-locked loop (PLL) configuration in your design. For example, you can view which control signals of the PLL are used with the settings for your PLL.

**Timing**—View the delay between the inputs and outputs of FPGA elements. For example, you can analyze the timing of the DATAB input to the COMBOUT output.

In addition, you can modify the following device properties with the Chip Planner:

- LEs and ALMs
- I/O cells
- PLLs
- Registers in RAM and DSP blocks
- Connections between elements
- Placement of elements

For more information about LEs, ALMs, and other resources of an FPGA device, refer to the relevant device handbook.

**Viewing Available Clock Networks in the Device**

When you select a task with clock region layer preset enabled, you can display the areas of the chip that are driven by global and regional clock networks.

Depending on the clock layers activated in the selected preset, the Chip Planner displays regional and global clock regions in the device, and the connectivity between clock regions, pins, and PLLs. Clock regions appear as rectangular overlay boxes with labels indicating the clock type and index. You can select each clock network region by clicking on the clock region. The clock-shaped icon at the top-left corner indicates that the region represents a clock network region. You can change the color in which the Chip Planner displays clock regions on the **Options** dialog box of the Tools menu.

The **Layers Settings** dialog box lists layers for different clock region types; when the selected device does not contain a given clock region, the option for that category is unavailable in the dialog box. You can customize the Chip Planner's display of clock regions by creating a custom preset with selected clock layers enabled in the **Layers Settings** dialog box.

**Viewing Critical Paths**

Critical paths are timing paths in your design that have a negative slack. These timing paths can span from device I/Os to internal registers, registers to registers, or from registers to device I/Os.

The slack of a path determines its criticality; slack appears in the timing analysis report. Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. The analytical capability of the Chip Planner helps you close timing on complex designs.

Viewing critical paths in the Chip Planner shows why a specific path is failing. You can see if any modification in the placement can reduce the negative slack. You can display details of a path (to expand/
collapse the path to/from the connections in the path) by clicking Expand Connections in the toolbar, or by clicking on the “+/-” on the label.

You can locate failing paths from the timing report in the TimeQuest Timing Analyzer. To locate the critical paths, click the Report Timing task from the Custom Reports group in the Tasks pane of the TimeQuest Timing Analyzer. From the View pane, which lists the failing paths, right-click on any failing path or node, and select Locate Path. From the Locate dialog box, select Chip Planner to see the failing path in the Chip Planner.

**Note:** To display paths in the floorplan, you must first make timing settings and perform a timing analysis.

**Related Information**
The Quartus Prime TimeQuest Timing Analyzer
For more information about performing static timing analysis with the TimeQuest Timing Analyzer.

**Viewing Routing Congestion**

The Report Routing Utilization task allows you to determine the percentage of routing resources in use following a compilation. This feature can identify where there is a lack of routing resources, helping you to make design changes to meet routing congestion design requirements.

2. To view the routing congestion in the Chip Planner, double-click the Report Routing Utilization command in the Tasks list.
3. Click Preview in the Report Routing Utilization dialog box to preview the default congestion display.
4. Change the Routing Utilization Type to display congestion for specific resources.

**Note:** The default display uses dark blue for 0% congestion (blue indicates zero utilization) and red for 100%. You can adjust the slider for Threshold percentage to change the congestion threshold level.

The routing congestion map uses the color and shading of logic resources to indicate relative resource utilization; darker shading represents a greater utilization of routing resources. Areas where routing utilization exceeds the threshold value specified in the Report Routing Utilization dialog box appear in red. The congestion map can help you determine whether you can modify the floorplan, or modify the RTL to reduce routing congestion.

To identify a lack of routing resources, it is necessary to investigate each routing interconnect type separately by selecting each interconnect type in turn in the Routing Utilization Settings dialog box.

The Compiler messages contain information about average and peak interconnect usage. Peak interconnect usage over 75%, or average interconnect usage over 60%, could be an indication that it might be difficult to fit your design. Similarly, peak interconnect usage over 90%, or average interconnect usage over 75%, are likely to have increased chances of not getting a valid fit.

**Viewing I/O Banks**
The Chip Planner shows all of the I/O banks of the device. To see the I/O bank map of the device, select Report All I/O Banks in the Tasks pane.

**Viewing High-Speed Serial Interfaces (HSSI)**
The Chip Planner displays a detailed block view of the receiver and transmitter channels of the high-speed serial interfaces. To display the HSSI block view, select Report HSSI Block Connectivity.
Generating Fan-In and Fan-Out Connections

The ability to display fan-in and fan-out connections enables you to view the atoms that fan-in to or fan-out from the selected atom. To remove the connections displayed, use the Clear Unselected Connections icon in the Chip Planner toolbar.

Generating Immediate Fan-In and Fan-Out Connections

The ability to display immediate fan-in and fan-out connections enables you to view the resource that is the immediate fan-in or fan-out connection for the selected atom. For example, if you select a logic resource and choose to view the immediate fan-in for that resource, you can see the routing resource that drives the logic resource. You can generate immediate fan-ins and fan-outs for all logic resources and routing resources. To remove the displayed connections from the screen, click the Clear Unselected Connections icon in the toolbar.

Highlight Routing

The Show Physical Routing command in the Locate History pane enables you to highlight the routing resources used by a selected path or connection.

Figure 12-1: Highlight Routing
Show Delays

With the Show Delays command, you can view timing delays for paths located from TimeQuest Timing Analyzer reports. For example, you can view the delay between two logic resources or between a logic resource and a routing resource.

![Figure 12-2: Show Delays Associated in a TimeQuest Timing Analyzer Path](image)

Exploring Paths in the Chip Planner

You can use the Chip Planner to explore paths between logic elements. The following example uses the Chip Planner to traverse paths from the Timing Analysis report.

Locate Path from the Timing Analysis Report to the Chip Planner

To locate a path from the Timing Analysis report to the Chip Planner, perform the following steps:

1. Select the path you want to locate in the Timing Analysis report.
2. Right-click the path and point to Locate Path > Locate in Chip Planner. The path is displayed with its timing data in the Chip Planner main window and is listed in the Locate History window.
3. To view the routing resources taken for a path you have located in the Chip Planner, use one of the following methods:

---

**Analyzing and Optimizing the Design Floorplan with the Chip Planner**

Alterna Corporation

Send Feedback
Analyzing Connections for a Path

To determine the connections between items in the Chip Planner, click the **Expand Connections** icon on the toolbar. To add the timing delays for paths you locate from the TimeQuest Timing Analyzer, click the **Show Delays** icon on the toolbar. To see the constituent delays on the selected path, click the “+” sign next to the path delay displayed in the Chip Planner.

![Figure 12-3: Path Analysis in the Chip Planner of a Path Located from TimeQuest](image)

Viewing Assignments in the Chip Planner

You can view location assignments by selecting the appropriate layer set in the Chip Planner. To view location assignments, select the **Floorplan Editing** preset or any custom preset that displays block utilization, and the Assignment editing mode. The Chip Planner shows location assignments graphically, by displaying assigned resources in a particular color (gray, by default). You can create or move an assignment by dragging the selected resource to a new location.
You can make node and pin location assignments to LogicLock Plus regions and custom regions using the drag-and-drop method in the Chip Planner. The Fitter applies the assignments that you create during the next place-and-route operation.

**Viewing High-Speed and Low-Power Tiles in the Chip Planner**

Some Altera devices have ALMs that can operate in either high-speed mode or low-power mode. The power mode is set during the fitting process in the Quartus Prime software. These ALMs are grouped together to form larger blocks, called “tiles.”

To view a power map, select **Tasks > Report High-Speed/Low-Power Tiles** after running the Fitter. The Chip Planner displays low-power and high-speed tiles in contrasting colors; yellow tiles operate in a high-speed mode, while blue tiles operate in a low-power mode. When you select the **Power** task, you can perform all floorplan-related functions for this task; however, you cannot edit tiles to change the power mode.

**Scripting Support**

You can run procedures and specify the settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt.

For detailed information about scripting command options, refer to the Quartus Prime command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```
Creating LogicLock Plus Assignments

The Quartus Prime software supports Tcl commands to create or modify LogicLock Plus assignments.

Create or Modify a Placement Region

The following assignment creates a new placement region with bounding box coordinates 10, 10, 20, 20. Use the same command format to modify an existing assignment.

```
set_instance_assignment -name PLACE_REGION -to <node name(s)> "10 10 20 20"
```

**Note:** To specify a non-rectangular or disjoint region, use a semicolon (;) as the delimiter between two or more bounding boxes.

Assign multiple instances to the same region with multiple PLACE_REGION instance assignments.

Create or Modify a Routing Region

The following assignment creates a routing region with bounding box coordinates 5, 5, 30, 30. Use the same command format to modify an existing assignment.

```
set_instance_assignment -name ROUTE_REGION -to <node name(s)> "5 5 30 30"
```

**Note:** All instances with a routing region assignment must have a respective placement region; the routing region must fully contain the placement region.

Routing regions cannot overlap.

Specify a Region as Reserved

The following assignment reserves an existing region:

```
set_instance_assignment -name <instance name> RESERVE_PLACE_REGION -to <node name(s)> ON
```

**Note:** You can only reserve placement regions.
Specify a Region as Core Only

By default, the Quartus Prime Pro Edition software includes pins in LogicLock Plus assignments. To specify a region as core only (i.e., periphery logic in the instance is not constrained), use the following assignment:

```plaintext
set_instance_assignment -name <instance name> CORE_ONLY_PLACE_REGION -to <node name(s)> ON
```

Assigning Virtual Pins

Use the following Tcl command to turn on the virtual pin setting for a pin called my_pin:

```plaintext
set_instance_assignment -name VIRTUAL_PIN ON -to my_pin
```

Related Information

Virtual Pins on page 12-6
Tcl Scripting on page 5-1
For more information about Tcl scripting.

Document Revision History

Table 12-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information on how to use LogicLock Plus regions.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Added information about color coding of LogicLock Plus regions.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>Updated description of Virtual Pins assignment to clarify that assigned input is not available.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device information.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>Updated “Viewing Routing Congestion” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated references to Quartus UI controls for the Chip Planner</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| May 2011     | 11.0.0  | • Updated for the 11.0 release.  
               |         |   Edited “LogicLock Plus Regions”  
               |         |   Updated “Viewing Routing Congestion”  
               |         |   Updated “Locate History”  
               |         |   Updated Figures 15-4, 15-9, 15-10, and 15-13  
               |         |   Added Figure 15-6 |
| December 2010| 10.1.0  | • Updated for the 10.1 release.  |
| July 2010    | 10.0.0  | • Updated device support information  
               |         |   Removed references to Timing Closure Floorplan; removed “Design Analysis Using the Timing Closure Floorplan” section  
               |         |   Added links to online Help topics  
               |         |   Added “Using LogicLock Plus Regions with the Design Partition Planner” section  
               |         |   Updated “Viewing Critical Paths” section  
               |         |   Updated several graphics  
               |         |   Updated format of Document revision History table |
| November 2009| 9.1.0   | • Updated supported device information throughout  
               |         |   Removed deprecated sections related to the Timing Closure Floorplan for older device families. (For information on using the Timing Closure Floorplan with older device families, refer to previous versions of the Quartus Prime Handbook, available in the Quartus Prime Handbook Archive.)  
               |         |   Updated “Creating Nonrectangular LogicLock Plus Regions” section  
               |         |   Added “Selected Elements Window” section  
               |         |   Updated table 12-1 |
May 2008 8.0.0  
- Updated the following sections:  
  “Chip Planner Tasks and Layers”  
  “LogicLock Plus Regions”  
  “Back-Annotating LogicLock Plus Regions”  
  “LogicLock Plus Regions in the Timing Closure Floorplan”  
- Added the following sections:  
  “Reserve LogicLock Plus Region”  
  “Creating Nonrectangular LogicLock Plus Regions”  
  “Viewing Available Clock Networks in the Device”  
- Updated Table 10–1  
- Removed the following sections:  
  Reserve LogicLock Plus Region Design Analysis Using the Timing Closure Floorplan

Related Information

Quartus Handbook Archive
For previous versions of the Quartus Prime Handbook, refer to the Quartus Handbook Archive.
Netlist Optimizations and Physical Synthesis

The Quartus Prime software offers additional physical synthesis optimizations that improve performance. The Compiler performs physical synthesis optimization during fitting.

Netlist optimizations operate with the atom netlist of your design, which describes a design in terms of specific primitives. An atom netlist file can be an Electronic Design Interchange Format (.edf) file or a Verilog Quartus Mapping (.vqm) file generated by a third-party synthesis tool. Quartus Prime synthesis generates and internally uses the atom netlist internally.

Physical synthesis optimizations are applied at different stages of the Quartus Prime compilation flow, either during synthesis, fitting, or both.

Note: Because the node names for primitives in the design can change when you use physical synthesis optimizations, you should evaluate whether your design flow requires fixed node names. If you use a verification flow that might require fixed node names, such as the SignalTap® II Logic Analyzer, formal verification, or the LogicLock Plus based optimization flow (for legacy devices), you must turn off physical synthesis options.

This chapter explains how physical synthesis optimizations in the Quartus Prime software can modify your design’s netlist to improve the quality of results. This chapter also provides guidelines for applying the various physical synthesis options, and information about preserving compilation results through back-annotation.

WYSIWYG Primitive Resynthesis

If you use a third-party tool to synthesize your design, use the Perform WYSIWYG primitive resynthesis option to apply optimizations to the synthesized netlist.

The Perform WYSIWYG primitive resynthesis option directs the Quartus Prime software to un-map the logic elements (LEs) in an atom netlist to logic gates, and then re-map the gates back to Altera-specific primitives. Third-party synthesis tools generate either an .edf or .vqm atom netlist file using Altera-specific primitives. When you turn on the Perform WYSIWYG primitive resynthesis option, the Quartus Prime software uses device-specific techniques during the re-mapping process. This feature re-maps the design using the Optimization Technique specified for your project (Speed, Area, or Balanced).

The Perform WYSIWYG primitive resynthesis option unmaps and remaps only logic cells, also referred to as LCELL or LE primitives, and regular I/O primitives (which may contain registers). Double data rate
(DDR) I/O primitives, memory primitives, digital signal processing (DSP) primitives, and logic cells in carry/cascade chains are not remapped. This process does not process logic specified in an encrypted .vqm file or an .edf file, such as third-party intellectual property (IP).

The **Perform WYSIWYG primitive resynthesis** option can change node names in the .vqm file or .edf file from your third-party synthesis tool, because the primitives in the atom netlist are broken apart and then re-mapped by the Quartus Prime software. The re-mapping process removes duplicate registers. Registers that are not removed retain the same name after re-mapping.

Any nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected during WYSIWYG primitive resynthesis. You can use the Assignment Editor to apply the **Netlist Optimizations** logic option. This option disables WYSIWYG resynthesis for parts of your design.

**Note:** Primitive node names are specified during synthesis. When netlist optimizations are applied, node names might change because primitives are created and removed. HDL attributes applied to preserve logic in third-party synthesis tools cannot be maintained because those attributes are not written into the atom netlist, which the Quartus Prime software reads.

If you use the Quartus Prime software to synthesize your design, you can use the **Preserve Register (preserve)** and **Keep Combinational Logic (keep)** attributes to maintain certain nodes in the design.

**Figure 13-1: Quartus Prime Flow for WYSIWYG Primitive Resynthesis**

![Diagram showing WYSIWYG primitive resynthesis process]

**Perform WYSIWYG Primitive Resynthesis**

**Note:** The **Perform WYSIWYG primitive resynthesis** option has no effect if you are using Quartus Prime synthesis to synthesize your design.

To turn on the **Perform WYSIWYG primitive resynthesis** option, click **Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis)**.

**Performing Physical Synthesis Optimizations**

The Quartus Prime Fitter places and routes the logic cells to ensure critical portions of logic are close together and use the fastest possible routing resources.

However, routing delays are often a significant part of the typical critical path delay. Physical synthesis optimizations take into consideration placement information, routing delays, and timing information. The Fitter then focuses timing-driven optimizations at those critical parts of the design. The tight integration of the synthesis and fitting processes is known as physical synthesis.
The following sections describe the physical synthesis optimizations available in the Quartus Prime software, and how they can help improve performance and fitting for the selected device.

Nodes or entities that have the Netlist Optimizations logic option set to Never Allow are not affected by physical synthesis algorithms. You can use the Assignment Editor to apply the Netlist Optimizations logic option. Use this option to disable physical synthesis optimizations for parts of your design.

Some physical synthesis options affect only registered logic, while others affect only combinational logic. Select options based on whether you want to keep the registers intact. For example, if your verification flow involves formal verification, you might want to keep the registers intact.

To choose physical synthesis optimization options, click Assignments > Settings > Compiler Settings > Advanced Settings (Fitter).

Related Information
Compiler Settings Page (Settings Dialog Box)

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).

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.

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.

Physical Synthesis Options

The Quartus Prime software provides physical synthesis optimization options to improve fitting results. To access these options, click Assignments > Settings > Compiler Settings > Advanced Settings (Fitter).
Table 13-1: Physical Synthesis Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netlist Optimizations</td>
<td>You can use the Assignment Editor to apply the Netlist Optimizations logic option. Use this option to disable physical synthesis optimizations for parts of your design.</td>
</tr>
<tr>
<td>Perform asynchronous signal pipelining</td>
<td>The Fitter performs automatic insertion of pipeline stages for asynchronous clear and asynchronous load signals during fitting when these signals negatively affect performance. You can use this option if asynchronous control signal recovery and removal times are not achieving requirements.</td>
</tr>
<tr>
<td>Perform physical synthesis for combinational logic</td>
<td>Swaps the look-up table (LUT) ports within LEs so that the critical path has fewer layers through which to travel. Also allows the duplication of LUTs to enable further optimizations on the critical path.</td>
</tr>
<tr>
<td>Perform Register Retiming for Performance</td>
<td>Enables the movement of registers across combinational logic, allowing the Quartus Prime software to trade off the delay between timing-critical paths and non-critical paths.</td>
</tr>
<tr>
<td>Physical Synthesis for Combinational Logic for Fitting</td>
<td>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.</td>
</tr>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>Specifies whether to perform WYSIWYG primitive resynthesis during synthesis. This option uses the setting specified in the Optimization Technique logic option.</td>
</tr>
<tr>
<td>Spectra-Q Physical Synthesis</td>
<td>Uses the Spectra-Q physical synthesis engine to perform combinational and sequential optimization during fitting to improve circuit performance.</td>
</tr>
</tbody>
</table>

Applying Netlist Optimization Options

The improvement in performance when using netlist optimizations is design dependent. If you have restructured your design to balance critical path delays, netlist optimizations might yield minimal improvement in performance.

You may have to experiment with available options to see which combination of settings works best for a particular design. Refer to the messages in the compilation report to see the magnitude of improvement with each option, and to help you decide whether you should turn on a given option or specific effort level.

Turning on more netlist optimization options can result in more changes to the node names in the design; bear this in mind if you are using a verification flow, such as the SignalTap II Logic Analyzer or formal verification that requires fixed or known node names.

To find the best results, you can use the Quartus Prime Design Space Explorer II (DSE) to apply various sets of netlist optimization options.
Viewing Synthesis and Netlist Optimization Reports

Physical synthesis optimizations performed during synthesis write results to the synthesis report. To access this report, perform the following steps:

2. In the Compilation Report list, open the Synthesis folder to view synthesis results.
3. In the Compilation Report list, open the Fitter folder to view the Netlist Optimizations table.

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:

```bash
quartus_sh --qhelp
```

You can specify many of the options described in this section on either an instance or global level, or both. Use the following Tcl command to make a global assignment:

```tcl
set_global_assignment -name <QSF variable name> <value>
```

Use the following Tcl command to make an instance assignment:

```tcl
set_instance_assignment -name <QSF variable name> <value> \
-to <instance name>
```

Related Information

- Tcl Scripting on page 5-1
- API Functions for Tcl
- Command-Line Scripting on page 4-1
- Quartus Prime Settings File Manual

Synthesis Netlist Optimizations

The project .qsf file preserves the settings that you specify in the GUI. Alternatively, you can edit the .qsf directly. The .qsf file supports the following synthesis netlist optimization commands. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.
Table 13-2: Synthesis Netlist Optimizations and Associated Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Quartus Prime Settings File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform WYSIWYG Primitive Resynthesis</td>
<td>ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP</td>
<td>ON, OFF</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Optimization Mode</td>
<td>OPTIMIZATION_MODE</td>
<td>BALANCED, HIGH PERFORMANCE EFFORT, AGGRESSIVE PERFORMANCE</td>
<td>Global, Instance</td>
</tr>
<tr>
<td>Power-Up Don’t Care</td>
<td>ALLOW_POWER_UP_DONT_CARE</td>
<td>ON, OFF</td>
<td>Global</td>
</tr>
</tbody>
</table>

Physical Synthesis Optimizations

The project .qsf file preserves the settings that you specify in the GUI. Alternatively, you can edit the .qsf file directly. The .qsf file supports the following synthesis netlist optimization commands. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 13-3: Physical Synthesis Optimizations and Associated Settings

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Quartus Prime Settings File Variable Name</th>
<th>Values</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectra-Q Physical Synthesis</td>
<td>SPECTRAQ_PHYSICAL_SYNTHESIS</td>
<td>ON, OFF</td>
<td>Global</td>
</tr>
<tr>
<td>Perform Register Retiming</td>
<td>PHYSICAL_SYNTHESIS_REGISTER_RETIMING</td>
<td>ON, OFF</td>
<td>Global</td>
</tr>
</tbody>
</table>

Document Revision History

Table 13-4: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Spectra-Q Physical Synthesis.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>• Updated location of Fitter Settings, Analysis &amp; Synthesis Settings,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Physical Synthesis Optimizations Settings to Compiler Settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated DSE II content.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device information.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Added links to Quartus Prime Help in several sections.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Referenced Documents section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reformatted Document Revision History</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Added information to “Physical Synthesis for Registers—Register Retiming”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information to “Applying Netlist Optimization Options”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Made minor editorial updates</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Was chapter 11 in the 8.1.0 release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated the “Physical Synthesis for Registers—Register Retiming”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated the “Performing Physical Synthesis Optimizations”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deleted Gate-Level Register Retiming section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated the referenced documents</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8½” × 11” page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Updated “Physical Synthesis Optimizations for Performance on page 11-9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Physical Synthesis Options for Fitting on page 11-16</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
Signal Integrity Analysis with Third-Party Tools

With the ever-increasing operating speed of interfaces in traditional FPGA design, the timing and signal integrity margins between the FPGA and other devices on the board must be within specification and tolerance before a single PCB is built.

If the board trace is designed poorly or the route is too heavily loaded, noise in the signal can cause data corruption, while overshoot and undershoot can potentially damage input buffers over time.

As FPGA devices are used in high-speed applications, signal integrity and timing margin between the FPGA and other devices on the printed circuit board (PCB) are important aspects to consider to ensure proper system operation. To avoid time-consuming redesigns and expensive board respins, the topology and routing of critical signals must be simulated. The high-speed interfaces available on current FPGA devices must be modeled accurately and integrated into timing models and board-level signal integrity simulations. The tools used in the design of an FPGA and its integration into a PCB must be “board-aware”—able to take into account properties of the board routing and the connected devices on the board.

The Quartus Prime software provides methodologies, resources, and tools to ensure good signal integrity and timing margin between Altera FPGA devices and other components on the board. Three types of analysis are possible with the Quartus Prime software:

- I/O timing with a default or user-specified capacitive load and no signal integrity analysis (default)
- The Quartus Prime Enable Advanced I/O Timing option utilizing a user-defined board trace model to produce enhanced timing reports from accurate “board-aware” simulation models
- Full board routing simulation in third-party tools using Altera-provided or generated Input/Output Buffer Information Specification (IBIS) or HSPICE I/O models

I/O timing using a specified capacitive test load requires no special configuration other than setting the size of the load. I/O timing reports from the Quartus Prime TimeQuest or the Quartus Prime Classic Timing Analyzer are generated based only on point-to-point delays within the I/O buffer and assume the presence of the capacitive test load with no other details about the board specified. The default size of the load is based on the I/O standard selected for the pin. Timing is measured to the FPGA pin with no signal integrity analysis details.

The Enable Advanced I/O Timing option expands the details in I/O timing reports by taking board topology and termination components into account. A complete point-to-point board trace model is defined and accounted for in the timing analysis. This ability to define a board trace model is an example of how the Quartus Prime software is “board-aware.”
In this case, timing and signal integrity metrics between the I/O buffer and the defined far end load are analyzed and reported in enhanced reports generated by the Quartus Prime TimeQuest Timing Analyzer.

Related Information

- I/O Management on page 2-1
  For more information about defining capacitive test loads or how to use the Enable Advanced I/O Timing option to configure a board trace model.

**Signal Integrity Simulations with HSPICE and IBIS Models**

The Quartus Prime software can export accurate HSPICE models with the built-in HSPICE Writer. You can run signal integrity simulations with these complete HSPICE models in Synopsys HSPICE. IBIS models of the FPGA I/O buffers are also created easily with the Quartus Prime IBIS Writer.

You can run signal integrity simulations with these complete HSPICE models in Synopsys HSPICE.

You can integrate IBIS models into any third-party simulation tool that supports them, such as the Mentor Graphics® Hyperlynx software. With the ability to create industry-standard model definition files quickly, you can build accurate simulations that can provide data to help improve board-level signal integrity.

The I/O’s IBIS and HSPICE model creation available in the Quartus Prime software can help prevent problems before a costly board respin is required. In general, creating and running accurate simulations is difficult and time consuming. The tools in the Quartus Prime software automate the I/O model setup and creation process by configuring the models specifically for your design. With these tools, you can set up and run accurate simulations quickly and acquire data that helps guide your FPGA and board design.

The information about signal integrity in this chapter refers to board-level signal integrity based on I/O buffer configuration and board parameters, not simultaneous switching noise (SSN), also known as ground bounce or \( V_{CC} \) sag. SSN is a product of multiple output drivers switching at the same time, causing an overall drop in the voltage of the chip’s power supply. This can cause temporary glitches in the specified level of ground or \( V_{CC} \) for the device.

This chapter is intended for FPGA and board designers and includes details about the concepts and steps involved in getting designs simulated and how to adjust designs to improve board-level timing and signal integrity. Also included is information about how to create accurate models from the Quartus Prime software and how to use those models in simulation software.

The information in this chapter is meant for those who are familiar with the Quartus Prime software and basic concepts of signal integrity and the design techniques and components in good PCB design. Finally, you should know how to set up simulations and use your selected third-party simulation tool.

Related Information

- **AN 315: Guidelines for Designing High-Speed FPGA PCBs**
  For more information about SSN and ways to prevent it.

- **Altera Signal Integrity Center**
  For information about basic signal integrity concepts and signal integrity details pertaining to Altera FPGA devices.
I/O Model Selection: IBIS or HSPICE

The Quartus Prime software can export two different types of I/O models that are useful for different simulation situations, IBIS models and HSPICE models.

IBIS models define the behavior of input or output buffers through the use of voltage-current (V-I) and voltage-time (V-t) data tables. HSPICE models, often referred to as HSPICE decks, include complete physical descriptions of the transistors and parasitic capacitances that make up an I/O buffer along with all the parameter settings required to run a simulation. The HSPICE decks generated by the Quartus Prime software are preconfigured with the I/O standard, voltage, and pin loading settings for each pin in your design.

The choice of I/O model type is based on many factors.

Table 14-1: IBIS and HSPICE Model Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>IBIS Model</th>
<th>HSPICE Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Buffer Description</td>
<td><strong>Behavioral</strong>—I/O buffers are described by voltage-current and voltage-time tables in typical, minimum, and maximum supply voltage cases.</td>
<td><strong>Physical</strong>—I/O buffers and all components in a circuit are described by their physical properties, such as transistor characteristics and parasitic capacitances, as well as their connections to one another.</td>
</tr>
<tr>
<td>Model Customization</td>
<td><strong>Simple and limited</strong>—The model completely describes the I/O buffer and does not usually have to be customized.</td>
<td><strong>Fully customizable</strong>—Unless connected to an arbitrary board description, the description of the board trace model must be customized in the model file. All parameters of the simulation are also adjustable.</td>
</tr>
<tr>
<td>Simulation Set Up and Run Time</td>
<td><strong>Fast</strong>—Simulations run quickly after set up correctly.</td>
<td><strong>Slow</strong>—Simulations take time to set up and take longer to run and complete.</td>
</tr>
<tr>
<td>Simulation Accuracy</td>
<td><strong>Good</strong>—For most simulations, accuracy is sufficient to make useful adjustments to the FPGA and/or board design to improve signal integrity.</td>
<td><strong>Excellent</strong>—Simulations are highly accurate, making HSPICE simulation almost a requirement for any high-speed design where signal integrity and timing margins are tight.</td>
</tr>
<tr>
<td>Third-Party Tool Support</td>
<td><strong>Excellent</strong>—Almost all third-party board simulation tools support IBIS.</td>
<td><strong>Good</strong>—Most third-party tools that support SPICE support HSPICE. However, Synopsys HSPICE is required for simulations of Altera’s encrypted HSPICE models.</td>
</tr>
</tbody>
</table>

Related Information

AN 283: Simulating Altera Devices with IBIS Models
For more information about IBIS files created by the Quartus Prime IBIS Writer and IBIS files in general, as well as links to websites with detailed information.
FPGA to Board Signal Integrity Analysis Flow

Board signal integrity analysis can take place at any point in the FPGA design process and is often performed before and after board layout. If it is performed early in the process as part of a pre-PCB layout analysis, the models used for simulations can be more generic.

These models can be changed as much as required to see how adjustments improve timing or signal integrity and help with the design and routing of the PCB. Simulations and the resulting changes made at this stage allow you to analyze “what if” scenarios to plan and implement your design better. To assist with early board signal integrity analysis, you can download generic IBIS model files for each device family and obtain HSPICE buffer simulation kits from the “Board Level Tools” section of the EDA Tool Support Resource Center.

Typically, if board signal integrity analysis is performed late in the design, it is used for a post-layout verification. The inputs and outputs of the FPGA are defined, and required board routing topologies and constraints are known. Simulations can help you find problems that might still exist in the FPGA or board design before fabrication and assembly. In either case, a simple process flow illustrates how to create accurate IBIS and HSPICE models from a design in the Quartus Prime software and transfer them to third-party simulation tools.

Your design depends on the type of model, IBIS or HSPICE, that you use for your simulations. When you understand the steps in the analysis flow, refer to the section of this chapter that corresponds to the model type you are using.
**Related Information**

**EDA Tool Support Resource Center**
For more information, generic IBIS model files for each device family, and to obtain HSPICE buffer simulation kits.

**Create I/O and Board Trace Model Assignments**

You can configure a board trace model for output signals or for bidirectional signals in output mode. You can then automatically transfer its description to HSPICE decks generated by the HSPICE Writer. This helps improve simulation accuracy.
To configure a board trace model, in the Settings dialog box, in the TimeQuest Timing Analyzer page, turn on the Enable Advanced I/O Timing option and configure the board trace model assignment settings for each I/O standard used in your design. You can add series or parallel termination, specify the transmission line length, and set the value of the far-end capacitive load. You can configure these parameters either in the Board Trace Model view of the Pin Planner, or click SettingsDeviceDevice and Pin Options.

The Quartus Prime software can generate IBIS models and HSPICE decks without having to configure a board trace model with the Enable Advanced I/O Timing option. In fact, IBIS models ignore any board trace model settings other than the far-end capacitive load. If any load value is set other than the default, the delay given by IBIS models generated by the IBIS Writer cannot be used to account correctly for the double counting problem. The load value mismatch between the IBIS delay and the $t_{CO}$ measurement of the Quartus Prime software prevents the delays from being safely added together. Warning messages displayed when the EDA Netlist Writer runs indicate when this mismatch occurs.

Related Information
- I/O Management on page 2-1
  For information about how to use the Enable Advanced I/O Timing option and configure board trace models for the I/O standards used in your design.

Output File Generation

IBIS and HSPICE model files are not generated by the Quartus Prime software by default. To generate or update the files automatically during each project compilation, select the type of file to generate and a location where to save the file in the project settings.

The IBIS and HSPICE Writers in the Quartus Prime software are run as part of the EDA Netlist Writer during normal project compilation. If either writer is turned on in the project settings, IBIS or HSPICE files are created and stored in the specified location. For IBIS, a single file is generated containing information about all assigned pins. HSPICE file generation creates separate files for each assigned pin. You can run the EDA Netlist Writer separately from a full compilation in the Quartus Prime software or at the command line.

Customize the Output Files

The files generated by either the IBIS or HSPICE Writer are text files that you can edit and customize easily for design or experimentation purposes.

IBIS files downloaded from the Altera website must be customized with the correct RLC values for the specific device package you have selected for your design. IBIS files generated by the IBIS Writer do not require this customization because they are configured automatically with the RLC values for your selected device. HSPICE decks require modification to include a detailed description of your board. With Enable Advanced I/O Timing turned on and a board trace model defined in the Quartus Prime software, generated HSPICE decks automatically include that model’s parameters. However, Altera recommends that you replace that model with a more detailed model that describes your board design more accurately. A default simulation included in the generated HSPICE decks measures delay between the FPGA and the far-end device. You can make additions or adjustments to the default simulation in the generated files to change the parameters of the default simulation or to perform additional measurements.

Set Up and Run Simulations in Third-Party Tools

When you have generated the files, you can use them to perform simulations in your selected simulation tool.
With IBIS models, you can apply them to input, output, or bidirectional buffer entities and quickly set up and run simulations. For HSPICE decks, the simulation parameters are included in the files. Open the files in Synopsys HSPICE and run simulations for each pin as required.

With HSPICE decks generated from the HSPICE Writer, the double counting problem is accounted for, which ensures that your simulations are accurate. Simulations that involve IBIS models created with anything other than the default loading settings in the Quartus Prime software must take the change in the size of the load between the IBIS delay and the Quartus Prime t<sub>CO</sub> measurement into account. Warning messages during compilation alert you to this change.

**Interpret Simulation Results**

If you encounter timing or signal integrity issues with your high-speed signals after running simulations, you can make adjustments to I/O assignment settings in the Quartus Prime software.

These could include such things as drive strength or I/O standard, or making changes to your board routing or topology. After regenerating models in the Quartus Prime software based on the changes you have made, rerun the simulations to check whether your changes corrected the problem.

**Simulation with IBIS Models**

IBIS models provide a way to run accurate signal integrity simulations quickly. IBIS models describe the behavior of I/O buffers with voltage-current and voltage-time data curves.

Because of their behavioral nature, IBIS models do not have to include any information about the internal circuit design of the I/O buffer. Most component manufacturers, including Altera, provide IBIS models for free download and use in signal integrity analysis simulation tools. You can download generic device family IBIS models from the Altera website for early design simulation or use the IBIS Writer to create custom IBIS models for your existing design.

**Elements of an IBIS Model**

An IBIS model file (.ibs) is a text file that describes the behavior of an I/O buffer across minimum, typical, and maximum temperature and voltage ranges with a specified test load.

The tables and values specified in the IBIS file describe five basic elements of the I/O buffer.

**Figure 14-2: Five Basic Elements of an I/O Buffer in IBIS Models**

The following elements correspond to each numbered block.
1. **Pulldown**—A voltage-current table describes the current when the buffer is driven low based on a pull-down voltage range of $-V_{CC}$ to 2 $V_{CC}$.

2. **Pullup**—A voltage-current table describes the current when the buffer is driven high based on a pull-up voltage range of $-V_{CC}$ to $V_{CC}$.

3. **Ground and Power Clamps**—Voltage-current tables describe the current when clamping diodes for electrostatic discharge (ESD) are present. The ground clamp voltage range is $-V_{CC}$ to $V_{CC}$, and the power clamp voltage range is $-V_{CC}$ to ground.

4. **Ramp and Rising/Falling Waveform**—A voltage-time ($dv/dt$) ratio describes the rise and fall time of the buffer during a logic transition. Optional rising and falling waveform tables can be added to more accurately describe the characteristics of the rising and falling transitions.

5. **Total Output Capacitance and Package RLC**—The total output capacitance includes the parasitic capacitances of the output pad, clamp diodes (if present), and input transistors. The package RLC is device package-specific and defines the resistance, inductance, and capacitance of the bond wire and pin of the I/O.

**Related Information**

**AN 283: Simulating Altera Devices with IBIS Models**

For more information about IBIS models and Altera-specific features, including links to the official IBIS specification.

**Creating Accurate IBIS Models**

There are two methods to obtain Altera device IBIS files for your board-level signal integrity simulations. You can download generic IBIS models from the Altera website. You can also use the IBIS writer in the Quartus Prime software to create design-specific models.

The IBIS file generated by the Quartus Prime software contains models of both input and output termination, and is supported for IBIS model versions of 4.2 and later. Arria V, Cyclone V, and Stratix V device families allow the use of bidirectional I/O with dynamic on-chip termination (OCT).

Dynamic OCT is used where a signal uses a series on-chip termination during output operation and a parallel on-chip termination during input operation. Typically this is used in Altera External Memory Interface IP.

The Quartus Prime IBIS dynamic OCT IBIS model names end in g50c_r50c. For example: sstl15i_ctnio_g50c_r50c.

In the simulation tool, the IBIS model is attached to a buffer.

- When the buffer is assigned as an output, use the series termination r50c.
- When the buffer is assigned as an input, use the parallel termination g50c.

**Download IBIS Models**

Altera provides IBIS models for almost all FPGA and FPGA configuration devices. You can use the IBIS models from the website to perform early simulations of the I/O buffers you expect to use in your design as part of a pre-layout analysis.

Downloaded IBIS models have the RLC package values set to one particular device in each device family.

The .ibs file can be customized for your device package and can be used for any simulation. IBIS models downloaded and used for simulations in this manner are generic. They describe only a certain set of models listed for each device on the Altera IBIS Models page of the Altera website. To create customized models for your design, use the IBIS Writer as described in the next section.
To simulate your design with the model accurately, you must adjust the RLC values in the IBIS model file to match the values for your particular device package by performing the following steps:

1. Download and expand the ZIP file (.zip) of the IBIS model for the device family you are using for your design. The .zip file contains the .ibs file along with an IBIS model user guide and a model data correlation report.
2. Download the Package RLC Values spreadsheet for the same device family.
3. Open the spreadsheet and locate the row that describes the device package used in your design.
4. From the package’s I/O row, copy the minimum, maximum, and typical values of resistance, inductance, and capacitance for your device package.
5. Open the .ibs file in a text editor and locate the [Package] section of the file.
6. Overwrite the listed values copied with the values from the spreadsheet and save the file.

Related Information
Altera IBIS Models
For information about whether models for your selected device are available.

Generate Custom IBIS Models with the IBIS Writer
If you have started your FPGA design and have created custom I/O assignments, you can use the Quartus Prime IBIS Writer to create custom IBIS models to accurately reflect your assignments.

Examples of custom assignments include drive strength settings or the enabling of clamping diodes for ESD protection. IBIS models created with the IBIS Writer take I/O assignment settings into account.

If the Enable Advanced I/O Timing option is turned off, the generated .ibs files are based on the load value setting for each I/O standard on the Capacitive Loading page of the Device and Pin Options dialog box in the Device dialog box. With the Enable Advanced I/O Timing option turned on, IBIS models use an effective capacitive load based on settings found in the board trace model on the Board Trace Model page in the Device and Pin Options dialog box or the Board Trace Model view in the Pin Planner. The effective capacitive load is based on the sum of the Near capacitance, Transmission line distributed capacitance, and the Far capacitance settings in the board trace model. Resistance values and transmission line inductance values are ignored.

Note: If you made any changes from the default load settings, the delay in the generated IBIS model cannot safely be added to the Quartus Prime t<sub>CO</sub> measurement to account for the double counting problem. This is because the load values between the two delay measurements do not match. When this happens, the Quartus Prime software displays warning messages when the EDA Netlist Writer runs to remind you about the load value mismatch.

Related Information
• Generating IBIS Output Files with the Quartus Prime Software
  For step-by-step instructions on how to generate IBIS models with the Quartus Prime software, refer to Quartus Prime Help.
• AN 283: Simulating Altera Devices with IBIS Models
  For more information about IBIS model generation.

Design Simulation Using the Mentor Graphics HyperLynx® Software
You must integrate IBIS models downloaded from the Altera website or created with the Quartus Prime IBIS Writer into board design simulations to accurately model timing and signal integrity.

The HyperLynx software from Mentor Graphics is one of the most popular tools for design simulation. The HyperLynx software makes it easy to integrate IBIS models into simulations.
The HyperLynx software is a PCB analysis and simulation tool for high-speed designs, consisting of two products, LineSim and BoardSim. LineSim is an early simulation tool. Before any board routing takes place, LineSim is used to simulate “what if” scenarios to assist in creating routing rules and defining board parameters. BoardSim is a post-layout tool used to analyze existing board routing. Specific nets are selected from a board layout file and simulated in a manner similar to LineSim. With board and routing parameters, and surrounding signal routing known, highly accurate simulations of the final fabricated PCB are possible. This section focuses on LineSim. Because the process of creating and running simulations is very similar for both LineSim and BoardSim, the details of IBIS model use in LineSim applies to simulations in BoardSim.

Simulations in LineSim are configured using a schematic GUI to create connections and topologies between I/O buffers, route trace segments, and termination components. LineSim provides two methods for creating routing schematics: cell-based and free-form. Cell-based schematics are based on fixed cells consisting of typical placements of buffers, trace impedances, and components. Parts of the grid-based cells are filled with the desired objects to create the topology. A topology in a cell-based schematic is limited by the available connections within and between the cells.

A more robust and expandable way to create a circuit schematic for simulation is to use the free-form schematic format in LineSim. The free-form schematic format makes it easy to place parts into any configuration and edit them as required. This section describes the use of IBIS models with free-form schematics, but the process is nearly identical for cell-based schematics.

**Figure 14-3: HyperLynx LineSim Free-Form Schematic Editor**
When you use HyperLynx software to perform simulations, you typically perform the following steps:

1. Create a new LineSim free-form schematic document and set up the board stackup for your PCB using the Stackup Editor. In this editor, specify board layer properties including layer thickness, dielectric constant, and trace width.

2. Create a circuit schematic for the net you want to simulate. The schematic represents all the parts of the routed net including source and destination I/O buffers, termination components, transmission line segments, and representations of impedance discontinuities such as vias or connectors.

3. Assign IBIS models to the source and destination I/O buffers to represent their behavior during operation.

4. Attach probes from the digital oscilloscope that is built in to LineSim to points in the circuit that you want to monitor during simulation. Typically, at least one probe is attached to the pin of a destination I/O buffer. For differential signals, you can attach a differential probe to both the positive and negative pins at the destination.

5. Configure and run the simulation. You can simulate a rising or falling edge and test the circuit under different drive strength conditions.

6. Interpret the results and make adjustments. Based on the waveforms captured in the digital oscilloscope, you can adjust anything in the circuit schematic to correct any signal integrity issues, such as overshoot or ringing. If necessary, you can make I/O assignment changes in the Quartus Prime software, regenerate the IBIS file with the IBIS Writer, and apply the updated IBIS model to the buffers in your HyperLynx software schematic.

7. Repeat the simulations and circuit adjustments until you are satisfied with the results. When the operation of the net meets your design requirements, implement changes to your I/O assignments in the Quartus Prime software and/or adjust your board routing constraints, component values, and placement to match the simulation.

Related Information
www.mentor.com
For more information about HyperLynx software, including schematic creation, simulation setup, model usage, product support, licensing, and training.

Configuring LineSim to Use Altera IBIS Models
You must configure LineSim to find and use the downloaded or generated IBIS models for your design. To do this, add the location of your .ibs file or files to the LineSim Model Library search path. Then you apply a selected model to a buffer in your schematic.

To add the Quartus Prime software’s default IBIS model location, <project directory>/board/ibis, to the HyperLynx LineSim model library search path, perform the following steps in LineSim:

1. From the Options menu, click Directories. The Set Directories dialog box appears. The Model-library file path(s) list displays the order in which LineSim searches file directories for model files.
2. Click **Edit**. A dialog box appears where you can add directories and adjust the order in which LineSim searches them.

**Figure 14-5: LineSim Select Directories Dialog Box**

3. Click **Add**

4. Browse to the default IBIS model location, `<project directory>/board/ibis`. Click **OK**.

5. Click **Up** to move the IBIS model directory to the top of the list. Click **Generate Model Index** to update LineSim’s model database with the models found in the added directory.

6. Click **OK**. The IBIS model directory for your project is added to the top of the Model-library file path(s) list.

7. To close the **Set Directories** dialog box, click **OK**.

**Integrating Altera IBIS Models into LineSim Simulations**

When the location for IBIS files has been set, you can assign the downloaded or generated IBIS models to the buffers in your schematic. To do this, perform the following steps:
1. Double-click a buffer symbol in your schematic to open the Assign Models dialog box. You can also click Assign Models from the buffer symbol’s right-click menu.

**Figure 14-6: LineSim Assign Model Dialog Box**

2. The pin of the buffer symbol you selected should be highlighted in the Pins list. If you want to assign a model to a different symbol or pin, select it from the list.

3. Click Select. The Select IC Model dialog box appears.
4. To filter the list of available libraries to display only IBIS models, select .IBS. Scroll through the Libraries list, and click the name of the library for your design. By default, this is <project name>.ibs.

5. The device for your design should be selected as the only item in the Devices list. If not, select your device from the list.

6. From the Signal list, select the name of the signal you want to simulate. You can also choose to select by device pin number.

7. Click OK. The Assign Models dialog box displays the selected .ibs file and signal.

8. If applicable to the signal you chose, adjust the buffer settings as required for the simulation.

9. Select and configure other buffer pins from the Pins list in the same manner.

10. Click OK when all I/O models are assigned.

Running and Interpreting LineSim Simulations

You can now run any desired simulations and make adjustments to the I/O assignments or simulation parameters as required.

For example, if you see too much overshoot in the simulated signal at the destination buffer after running a simulation, you could adjust the drive strength I/O assignment setting to a lower value. Regenerate the .ibs file, and run the simulation again to verify whether the change fixed the problem.
If you see a discontinuity or other anomalies at the destination, such as slow rise and fall times, adjust the termination scheme or termination component values. After making these changes, rerun the simulation to check whether your adjustments solved the problem. In this case, it is not necessary to regenerate the .ibs file.

Related Information

Altera Signal Integrity Center
For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your design.
Simulation with HSPICE Models

HSPICE decks are used to perform highly accurate simulations by describing the physical properties of all aspects of a circuit precisely. HSPICE decks describe I/O buffers, board components, and all of the connections between them, as well as defining the parameters of the simulation to be run.

By their nature, HSPICE decks are highly customizable and require a detailed description of the circuit under simulation. For devices that support advanced I/O timing, when Enable Advanced I/O Timing is turned on, the HSPICE decks generated by the Quartus Prime HSPICE Writer automatically include board components and topology defined in the Board Trace Model. Configure the board components and topology in the Pin Planner or in the Board Trace Model tab of the Device and Pin Options dialog box. All HSPICE decks generated by the Quartus Prime software include compensation for the double count problem. You can simulate with the default simulation parameters built in to the generated HSPICE decks or make adjustments to customize your simulation.

Related Information
The Double Counting Problem in HSPICE Simulations on page 14-17

Supported Devices and Signaling

The HSPICE Writer in the Quartus Prime software supports Arria, Cylcone, and Stratix devices for the creation of a board trace model in the Quartus Prime software for automatic inclusion in an HSPICE deck.

The HSPICE files include the board trace description you create in the Board Trace Model view in the Pin Planner or the Board Trace Model tab in the Device and Pin Options dialog box.

Note: Note that for Arria 10 devices, you may need to download the Encrypted HSPICE model from the Altera website.

Related Information
I/O Management on page 2-1
For more information about the Enable Advanced I/O Timing option and configuring board trace models for the I/O standards in your design.

SPICE Models for Altera Devices
For more information about the Encrypted HSPICE model.

Accessing HSPICE Simulation Kits

You can access the available HSPICE models with the Quartus Prime software’s HSPICE Writer tool and also at the Spice Models for Altera Devices web page.

The Quartus Prime software HSPICE Writer tool removes many common sources of user error from the I/O simulation process. The HSPICE Writer tool automatically creates preconfigured I/O simulation spice decks that only require the addition of a user board model. All the difficult tasks required to configure the I/O modes and interpret the timing results are handled automatically by the HSPICE Writer tool.

Related Information
Spice Models for Altera Devices
For more information about downloadable HSPICE models.
The Double Counting Problem in HSPICE Simulations

Simulating I/Os using accurate models is extremely helpful for finding and fixing FPGA I/O timing and board signal integrity issues before any boards are built. However, the usefulness of such simulations is directly related to the accuracy of the models used and whether the simulations are set up and performed correctly. To ensure accuracy in models and simulations created for FPGA output signals, the timing hand-off between $t_{CO}$ timing in the Quartus Prime software and simulation-based board delay must be taken into account. If this hand-off is not handled correctly, the calculated delay could either count some of the delay twice or even miss counting some of the delay entirely.

Defining the Double Counting Problem

The double counting problem is inherent to the method output timing is analyzed versus the method used for HSPICE models. The timing analyzer tools in the Quartus Prime software measure delay timing for an output signal from the core logic of the FPGA design through the output buffer ending at the FPGA pin with a default capacitive load or a specified value for the selected I/O standard. This measurement is the $t_{CO}$ timing variable.

Figure 14-10: Double Counting Problem

HSPICE models for board simulation measure $t_{PD}$ (propagation delay) from an arbitrary reference point in the output buffer, through the device pin, out along the board routing, and ending at the signal destination.

It is apparent immediately that if these two delays were simply added together, the delay between the output buffer and the device pin would be counted twice in the calculation. A model or simulation that does not account for this double count would create overly pessimistic simulation results, because the double-counted delay can limit I/O performance artificially. To fix the problem, it might seem that simply subtracting the overlap between $t_{CO}$ and $t_{PD}$ would account for the double count. However, this adjustment would not be accurate because each measurement is based on a different load.

Note: Input signals do not exhibit this problem because the HSPICE models for inputs stop at the FPGA pin instead of at the input buffer. In this case, simply adding the delays together produces an accurate measurement of delay timing.
The Solution to Double Counting

To adjust the measurements to account for the double-counting, the delay between the arbitrary point in the output buffer selected by the HSPICE model and the FPGA pin must be subtracted from either $t_{CO}$ or $t_{PD}$ before adding the results together. The subtracted delay must also be based on a common load between the two measurements. This is done by repeating the HSPICE model measurement, but with the same load used by the Quartus Prime software for the $t_{CO}$ measurement.

**Figure 14-11: Common Test Loads Used for Output Timing**

![Diagram showing the timing of signals from FPGA logic to signal destination on the board, including HSPICE netlist with Quartus Prime test load, HSPICE netlist with user board trace model, Quartus Prime $t_{CO}$, HSPICE $t_{PD}$ with user specified board trace model, overlap (HSPICE delay with test load), HSPICE $t_{PD}$ adjusted by $t_{TESTLOAD}$, total delay, termination network/trace model, and signal destination.]

With $t_{TESTLOAD}$ known, the total delay is calculated for the output signal from the FPGA logic to the signal destination on the board, accounting for the double count.

\[ t_{delay} = t_{CO} + (t_{PD} - t_{TESTLOAD}) \]

The preconfigured simulation files generated by the HSPICE Writer in the Quartus Prime software are designed to account for the double-counting problem based on this calculation automatically. Performing accurate timing simulations is easy without having to make adjustments for double counting manually.

**HSPICE Writer Tool Flow**

This section includes information to help you get started using the Quartus Prime software HSPICE Writer tool. The information in this section assumes you have a basic knowledge of the standard Quartus Prime software design flow, such as project and assignment creation, compilation, and timing analysis.

**Related Information**

**Quartus Prime Handbook**

For additional information about standard design flows.
Applying I/O Assignments

The first step in the HSPICE Writer tool flow is to configure the I/O standards and modes for each of the pins in your design properly. In the Quartus Prime software, these settings are represented by assignments that map I/O settings, such as pin selection, and I/O standard and drive strength, to corresponding signals in your design.

The Quartus Prime software provides multiple methods for creating these assignments:

- Using the Pin Planner
- Using the assignment editor
- Manually editing the .qsf file
- By making assignments in a scripted Quartus Prime flow using Tcl

Enabling HSPICE Writer

You must enable the HSPICE Writer in the Settings dialog box of the Quartus Prime software to generate the HSPICE decks from the Quartus Prime software.

Figure 14-12: EDA Tool Settings: Board Level Options Dialog Box

Enabling HSPICE Writer Using Assignments

You can also use HSPICE Writer in conjunction with a scripted Tcl flow. To enable HSPICE Writer during a full compile, include the following lines in your Tcl script.

Enable HSPICE Writer

```tcl
set_global_assignment -name EDA_BOARD_DESIGN_SIGNAL_INTEGRITY_TOOL "HSPICE (Signal Integrity)"
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT HSPICE -section_id eda_board_design_signal_integrity
set_global_assignment -name EDA_NETLIST_WRITER_OUTPUT_DIR <output_directory> -section_id eda_board_design_signal_integrity
```
As with command-line invocation, specifying the output directory is optional. If not specified, the output directory defaults to **board/hspice**.

**Naming Conventions for HSPICE Files**

HSPICE Writer automatically generates simulation files and names them using the following naming convention: `<device>_<pin #>_<pin_name>_<in/out>.sp`.

For bidirectional pins, two spice decks are produced; one with the I/O buffer configured as an input, and the other with the I/O buffer configured as an output.

The Quartus Prime software supports alphanumeric pin names that contain the underscore (\_) and dash (-) characters. Any illegal characters used in file names are converted automatically to underscores.

**Related Information**

- Sample Output for I/O HSPICE Simulation Deck on page 14-30
- Sample Input for I/O HSPICE Simulation Deck on page 14-26

**Invoking HSPICE Writer**

After HSPICE Writer is enabled, the HSPICE simulation files are generated automatically each time the project is completely compiled. The Quartus Prime software also provides an option to generate a new set of simulation files without having to recompile manually. In the Processing menu, click **Start EDA Netlist Writer** to generate new simulation files automatically.

**Note:** You must perform both Analysis & Synthesis and Fitting on a design before invoking the HSPICE Writer tool.

**Invoking HSPICE Writer from the Command Line**

If you use a script-based flow to compile your project, you can create HSPICE model files by including the following commands in your Tcl script (.tcl file).

Create HSPICE Model Files

```tcl
set_global_assignment -name EDA_BOARD_DESIGN_SIGNAL_INTEGRITY_TOOL "HSPICE (Signal Integrity)"
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT HSPICE
-section_ideda_board_design_signal_integrity
set_global_assignment -name EDA_NETLIST_WRITER_OUTPUT_DIR <output_directory>
-section_id eda_board_design_signal_integrity
```

The `<output_directory>` option specifies the location where HSPICE model files are saved. By default, the `<project directory>/board/hspice` directory is used.

Invoke HSPICE Writer

To invoke the HSPICE Writer tool through the command line, type:

```
quartus_eda.exe <project_name> --board_signal_integrity=on --format=HSPICE
--output_directory=<output_directory>
```

 `<output_directory>` specifies the location where the generated spice decks will be written (relative to the design directory). This is an optional parameter and defaults to **board/hspice**.

**Customizing Automatically Generated HSPICE Decks**

HSPICE models generated by the HSPICE Writer can be used for simulation as generated.
A default board description is included, and a default simulation is set up to measure rise and fall delays for both input and output simulations, which compensates for the double counting problem. However, Altera recommends that you customize the board description to more accurately represent your routing and termination scheme.

The sample board trace loading in the generated HSPICE model files must be replaced by your actual trace model before you can run a correct simulation. To do this, open the generated HSPICE model files for all pins you want to simulate and locate the following section.

**Sample Board Trace Section**

```plaintext
* I/O Board Trace and Termination Description
* - Replace this with your board trace and termination description
```

You must replace the example load with a load that matches the design of your PCB board. This includes a trace model, termination resistors, and, for output simulations, a receiver model. The spice circuit node that represents the pin of the FPGA package is called `pin`. The node that represents the far pin of the external device is called `load-in` (for output SPICE decks) and `source-in` (for input SPICE decks).

For an input simulation, you must also modify the stimulus portion of the spice file. The section of the file that must be modified is indicated in the following comment block.

**Sample Source Stimulus Section**

```plaintext
* Sample source stimulus placeholder
* - Replace this with your I/O driver model
```

Replace the sample stimulus model with a model for the device that will drive the FPGA.

**Running an HSPICE Simulation**

Because simulation parameters are configured directly in the HSPICE model files, running a simulation requires only that you open an HSPICE file in the HSPICE user interface and start the simulation.

**Figure 14-13: HSPICE User Interface Window**

![HSPICE User Interface Window](image)

Click **Open** and browse to the location of the HSPICE model files generated by the Quartus Prime HSPICE Writer. The default location for HSPICE model files is `<project directory>/board/hspice`. Select the `.sp` file generated by the HSPICE Writer for the signal you want to simulate. Click **OK**.

To run the simulation, click **Simulate**. The status of the simulation is displayed in the window and saved in an `.lis` file with the same name as the `.sp` file when the simulation is complete. Check the `.lis` file if an error occurs during the simulation requiring a change in the `.sp` file to fix.
Interpreting the Results of an Output Simulation

By default, the automatically generated output simulation spice decks are set up to measure three delays for both rising and falling transitions. Two of the measurements, \( t_{pd\_rise} \) and \( t_{pd\_fall} \), measure the double-counting corrected delay from the FPGA pin to the load pin. To determine the complete clock-edge to load-pin delay, add these numbers to the Quartus Prime software reported default loading \( t_{CO} \) delay.

The remaining four measurements, \( t_{pd\_uncomp\_rise}, t_{pd\_uncomp\_fall}, t_{dblcnt\_rise}, \) and \( t_{dblcnt\_fall} \), are required for the double-counting compensation process and are not required for further timing usage.

Related Information

Simulation Analysis on page 14-30

Interpreting the Results of an Input Simulation

By default, the automatically generated input simulation SPICE decks are set up to measure delays from the source’s driver pin to the FPGA’s input pin for both rising and falling transitions.

The propagation delay is reported by HSPICE measure statements as \( t_{pd\_rise} \) and \( t_{pd\_fall} \). To determine the complete source driver pin-to-FPGA register delay, add these numbers to the Quartus Prime software reported \( T_H \) and \( T_{SU} \) input timing numbers.

Viewing and Interpreting Tabular Simulation Results

The \( .lis \) file stores the collected simulation data in tabular form. The default simulation configured by the HSPICE Writer produces delay measurements for rising and falling transitions on both input and output simulations.

These measurements are found in the \( .lis \) file and named \( t_{pd\_rise} \) and \( t_{pd\_fall} \). For output simulations, these values are already adjusted for the double count. To determine the complete delay from the FPGA logic to the load pin, add either of these measurements to the Quartus Prime \( t_{CO} \) delay. For input simulations, add either of these measurements to the Quartus Prime \( t_{SU} \) and \( t_H \) delay values to calculate the complete delay from the far end stimulus to the FPGA logic. Other values found in the \( .lis \) file, such as \( t_{pd\_uncomp\_rise}, t_{pd\_uncomp\_fall}, t_{dblcnt\_rise}, \) and \( t_{dblcnt\_fall} \), are parts of the double count compensation calculation. These values are not necessary for further analysis.

Viewing Graphical Simulation Results

You can view the results of the simulation quickly as a graphical waveform display using the AvanWaves viewer included with HSPICE. With the default simulation configured by the HSPICE Writer, you can view the simulated waveforms at both the source and destination in input and output simulations.

To see the waveforms for the simulation, in the HSPICE user interface window, click **AvanWaves**. The AvanWaves viewer opens and displays the **Results Browser**.
The **Results Browser** lets you select which waveform to view quickly in the main viewing window. If multiple simulations are run on the same signal, the list at the top of the **Results Browser** displays the results of each simulation. Click the simulation description to select which simulation to view. By default, the descriptions are derived from the first line of the HSPICE file, so the description might appear as a line of asterisks.

Select the type of waveform to view, by performing the following steps:

1. To see the source and destination waveforms with the default simulation, from the **Types** list, select **Voltages**.
2. On the **Curves** list, double-click the waveform you want to view. The waveform appears in the main viewing window.

You can zoom in and out and adjust the view as desired.
Making Design Adjustments Based on HSPICE Simulations

Based on the results of your simulations, you can make adjustments to the I/O assignments or simulation parameters if required. For example, after you run a simulation and see overshoot or ringing in the simulated signal at the destination buffer, you can adjust the drive strength I/O assignment setting to a lower value. Regenerate the HSPICE deck, and run the simulation again to verify that the change fixed the problem.
If there is a discontinuity or any other anomalies at the destination, adjust the board description in the Quartus Prime Board Trace Model, or in the generated HSPICE model files to change the termination scheme or adjust termination component values. After making these changes, regenerate the HSPICE files if necessary, and rerun the simulation to verify whether your adjustments solved the problem.
Related Information

Altera Signal Integrity Center
For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your FPGA design.

Sample Input for I/O HSPICE Simulation Deck

The following sections examine a typical HSPICE simulation spice deck for an I/O of type input. Each section presents the simulation file one block at a time.

Header Comment

The first block of an input simulation spice deck is the header comment. The purpose of this block is to provide an easily readable summary of how the simulation file has been automatically configured by the Quartus Prime software.

This block has two main components: The first component summarizes the I/O configuration relevant information such as device, speed grade, and so on. The second component specifies the exact test condition that the Quartus Prime software assumes for the given I/O standard.

Sample Header Comment Block

* Quartus Prime HSPICE Writer I/O Simulation Deck*
* This spice simulation deck was automatically generated by Quartus for the following IO settings:
Simulation Conditions

The simulation conditions block loads the appropriate process corner models for the transistors. This condition is automatically set up for the slow timing corner and is modified only if other simulation corners are desired.

Simulation Conditions Block

* Process Settings

.join brief
.inc 'sii_tt.inc' * TT process corner

Simulation Options

The simulation options block configures the simulation temperature and configures HSPICE with typical simulation options.

Simulation Options Block

* Simulation Options

.options brief=0
.options badchr co=132 scale=1e-6 acct ingold=2 nomod dv=1.0
+      dcdstep=1 absv=1e-3 absi=1e-8 probe csdf=2 accurate=1
+      converge=1
+ .temp 85

Note: For a detailed description of these options, consult your HSPICE manual.
**Constant Definition**

The constant definition block of the simulation file instantiates the voltage sources that controls the configuration modes of the I/O buffer.

**Constant Definition Block**

```
* Constant Definition
  voeb       oeb       0     vc * Set to 0 to enable buffer output
  vopdrain   opdrain   0     0   * Set to vc to enable open drain
  vrambh     rambh     0     0   * Set to vc to enable bus hold
  vrpullup   rpullup   0     0   * Set to vc to enable weak pullup
  vpcdp5     rpcdp5    0     rp5 * Set the IO standard
  vpcdp4     rpcdp4    0     rp4
  vpcdp3     rpcdp3    0     rp3
  vpcdp2     rpcdp2    0     rp2
  vpcdp1     rpcdp1    0     rp1
  vpcdp0     rpcdp0    0     rp0
  vpcdn4     rpcdn4    0     rn4
  vpcdn3     rpcdn3    0     rn3
  vpcdn2     rpcdn2    0     rn2
  vpcdn1     rpcdn1    0     rn1
  vpcdn0     rpcdn0    0     rn0
  vdin      din       0     0

Where:
- Voltage source voeb controls the output enable of the buffer and is set to disabled for inputs.
- vopdrain controls the open drain mode for the I/O.
- vrambh controls the bus hold circuitry in the I/O.
- vrpullup controls the weak pullup.
- The next 11 voltages sources control the I/O standard of the buffer and are configured through a later library call.
- vdin is not used on input pins because it is the data pin for the output buffer.
```

**Buffer Netlist**

The buffer netlist block of the simulation spice deck loads all the load models required for the corresponding input pin.

**Buffer Netlist Block**

```
* IO Buffer Netlist
  .include 'vio_buffer.inc'
```

**Drive Strength**

The drive strength block of the simulation SPICE deck loads the configuration bits necessary to configure the I/O into the proper I/O standard and drive strengths.

Although these settings are not relevant to an input buffer, they are provided to allow the SPICE deck to be modifiable to support bidirectional simulations.
Drive Strength Block

* Drive Strength Settings

.lib ‘drive_select_hio.lib’ 3p3ttl_12ma

I/O Buffer Instantiation

The I/O buffer instantiation block of the simulation SPICE deck instantiates the necessary power supplies and I/O model components that are necessary to simulate the given I/O.

I/O Buffer Instantiation

I/O Buffer Instantiation

* Supply Voltages Settings

.param vcn=3.135
.param vpd=2.97
.param vc=1.15

* Instantiate Power Supplies

vvcc       vcc       0     vc     * FPGA core voltage
vvss       vss       0     0      * FPGA core ground
vvccn      vccn      0     vcn    * IO supply voltage
vvssn      vssn      0     0      * IO ground
vvccpd     vccpd     0     vpd    * Pre-drive supply voltage

* Instantiate I/O Buffer

xvio_buf din oeb opdrain die rambh
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp5 rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup vccn vccpd vcpad0 vio_buf

* Internal Loading on Pad
* - No loading on this pad due to differential buffer/support
* circuitry

* I/O Buffer Package Model
* - Single-ended I/O standard on a Row I/O
.lib ‘lib/package.lib’ hio
xpkg die pin hio_pkg

Board Trace and Termination

The board trace and termination block of the simulation SPICE deck is provided only as an example. Replace this block with your own board trace and termination models.

Board Trace and Termination Block

* I/O Board Trace and Termination Description
* - Replace this with your board trace and termination description

wtline pin vssn load vssn N=1 L=1 RLGCMODEL=tlinemodel
.MODEL tlinemodel W MODELTYPE=RLGC N=1 Lo=7.13n Co=2.85p
Rterm2 load vssn 1x
Stimulus Model

The stimulus model block of the simulation spice deck is provided only as a place holder example. Replace this block with your own stimulus model. Options for this include an IBIS or HSPICE model, among others.

Stimulus Model Block

* Sample source stimulus placeholder
* Replace this with your I/O driver model

\[ V_{source} \text{ source 0 pulse}(0 \text{ vcn 0}s \ 0.4ns \ 0.4ns \ 8.5ns \ 17.4ns) \]

Simulation Analysis

The simulation analysis block of the simulation file is configured to measure the propagation delay from the source to the FPGA pin. Both the source and end point of the delay are referenced against the 50% \( V_{CCN} \) crossing point of the waveform.

Simulation Analysis Block

* Simulation Analysis Setup
* Print out the voltage waveform at both the source and the pin
  \[ \text{.print tran } v(\text{source}) \ v(\text{pin}) \]
  \[ \text{.tran 0.020ns 17ns} \]
* Measure the propagation delay from the source pin to the pin
* referenced against the 50% voltage threshold crossing point
  \[ \text{.measure TRAN tpd\_rise TRIG v(source) val='vcn*0.5' rise=1} \]
  \[ + \ TARG v(pin) \ val = 'vcn*0.5' \ rise=1 \]
  \[ \text{.measure TRAN tpd\_fall TRIG v(source) val='vcn*0.5' fall=1} \]
  \[ + \ TARG v(pin) \ val = 'vcn*0.5' \ fall=1 \]

Sample Output for I/O HSPICE Simulation Deck

A typical HSPICE simulation SPICE deck for an I/O-type output has several sections. Each section presents the simulation file one block at a time.

Header Comment

The first block of an output simulation SPICE deck is the header comment. The purpose of this block is to provide a readable summary of how the simulation file has been automatically configured by the Quartus Prime software.

This block has two main components:

- The first component summarizes the I/O configuration relevant information such as device, speed grade, and so on.
- The second component specifies the exact test condition that the Quartus Prime software assumes when generating \( t_{CO} \) delay numbers. This information is used as part of the double-counting correction circuitry contained in the simulation file.

The SPICE decks are preconfigured to calculate the slow process corner delay but can also be used to simulate the fast process corner as well. The fast corner conditions are listed in the header under the notes section.
The final section of the header comment lists any warning messages that you must consider when you use the SPICE decks.

Header Comment Block

* Quartus Prime HSPICE Writer I/O Simulation Deck
* This spice simulation deck was automatically generated by
  Quartus Prime for the following IO settings:
* Device: EP2S60F1020C3
* Speed Grade: C3
* Pin: AA4 (out96)
* Bank: IO Bank 6 (Row I/O)
* I/O Standard: LVTTL, 12mA
* OCT: Off
* Quartus’ default I/O timing delays assume the following slow
  corner simulation conditions.
  Specified Test Conditions For Quartus Prime Tco
  Temperature: 85C (Slowest Temperature Corner)
  Transistor Model: TT (Typical Transistor Corner)
  Vccn: 3.135V (Vccn_min = Nominal - 5%)
  Vccpd: 2.97V (Vccpd_min = Nominal - 10%)
  Load: No Load
  Vtt: 1.5675V (Voltage reference is Vccn/2)
* For C3 devices, the TT transistor corner provides an
  approximation for worst case timing. However, for functionality
  simulations, it is recommended that the SS corner be simulated
  as well.
* Note: The I/O transistors are specified to operate at least as
  fast as the TT transistor corner, actual production
  devices can be as fast as the FF corner. Any simulations
  for hold times should be conducted using the fast process
  corner with the following simulation conditions.
  Temperature: 0C (Fastest Commercial Temperature Corner **)
  Transistor Model: FF (Fastest Transistor Corner)
  Vccn: 1.98V (Vccn_hold = Nominal + 10%)
  Vccpd: 3.63V (Vccpd_hold = Nominal + 10%)
  Vtt: 0.95V (Vtt_hold = Vccn/2 - 40mV)
  Vcc: 1.25V (Vcc_hold = Maximum Recommended)
  Package Model: Short-circuit from pad to pin
* Warnings:

Simulation Conditions

The simulation conditions block loads the appropriate process corner models for the transistors. This condition is automatically set up for the slow timing corner and must be modified only if other simulation corners are desired.

Simulation Conditions Block

* Process Settings
  .options brief
  .inc 'sii_tt.inc' * typical-typical process corner

Note: Two separate corners cannot be simulated at the same time. Instead, simulate the base case using the Quartus corner as one simulation and then perform a second simulation using the desired customer corner. The results of the two simulations can be manually added together.
Simulation Options

The simulation options block configures the simulation temperature and configures HSPICE with typical simulation options.

Simulation Options Block

* Simulation Options
  .options brief=0
  .options badchr co=132 scale=1e-6 acct ingold=2 nomod dv=1.0
  +        dcstep=1 absv=1e-3 absi=1e-8 probe csdf=2 accurate=1
  +        converge=1
  .temp 85

Note: For a detailed description of these options, consult your HSPICE manual.

Constant Definition

The constant definition block of the output simulation SPICE deck instantiates the voltage sources that controls the configuration modes of the I/O buffer.

Constant Definition Block

* Constant Definition

  voeb oeb 0 0 * Set to 0 to enable buffer output
  vopdrain opdrain 0 0 * Set to vc to enable open drain
  vrambh rambh 0 0 * Set to vc to enable bus hold
  vrpullup rpullup 0 0 * Set to vc to enable weak pullup
  vpci rpci 0 0 * Set to vc to enable pci mode
  vpcdp4 rpcdp4 0 rp4 * These control bits set the I/O standard
  vpcdp3 rpcdp3 0 rp3
  vpcdp2 rpcdp2 0 rp2
  vpcdp1 rpcdp1 0 rp1
  vpcdp0 rpcdp0 0 rp0
  vpcdn4 rpcdn4 0 rn4
  vpcdn3 rpcdn3 0 rn3
  vpcdn2 rpcdn2 0 rn2
  vpcdn1 rpcdn1 0 rn1
  vpcdn0 rpcdn0 0 rn0
  vdin din 0 pulse(0 vc 0s 0.2ns 0.2ns 8.5ns 17.4ns)

Where:

- Voltage source voeb controls the output enable of the buffer.
- vopdrain controls the open drain mode for the I/O.
- vrambh controls the bus hold circuitry in the I/O.
- vrpullup controls the weak pullup.
- vpci controls the PCI clamp.
- The next ten voltage sources control the I/O standard of the buffer and are configured through a later library call.
- vdin is connected to the data input of the I/O buffer.
- The edge rate of the input stimulus is automatically set to the correct value by the Quartus Prime software.

I/O Buffer Netlist

The I/O buffer netlist block loads all of the models required for the corresponding pin. These include a model for the I/O output buffer, as well as any loads that might be present on the pin.
I/O Buffer Netlist Block

*IO Buffer Netlist
.include 'hio_buffer.inc'
.include 'lvds_input_load.inc'
.include 'lvds_oct_load.inc'

Drive Strength
The drive strength block of the simulation spice deck loads the configuration bits for configuring the I/O to the proper I/O standard and drive strength. These options are set by the HSPICE Writer tool and are not changed for expected use.

Drive Strength Block
* Drive Strength Settings
.lib 'drive_select_hio.lib' 3p3ttl_12ma

Slew Rate and Delay Chain
Stratix and Cyclone devices have sections for configuring the slew rate and delay chain settings.

Slew Rate and Delay Chain Settings
* Programmable Output Delay Control Settings
.lib 'lib/output_delay_control.lib' no_delay
* Programmable Slew Rate Control Settings
.lib 'lib/slew_rate_control.lib' slow_slow

I/O Buffer Instantiation
The I/O buffer instantiation block of the output simulation spice deck instantiates the necessary power supplies and I/O model components that are necessary to simulate the given I/O.

I/O Buffer Instantiation Block
* I/O Buffer Instantiation

* Supply Voltages Settings
.param vcn=3.135
.param vpd=2.97
.param vc=1.15

* Instantiate Power Supplies
vvcc vcc 0 vc  * FPGA core voltage
vvss vss 0 0  * FPGA core ground
vvccn vccn 0 vcn  * IO supply voltage
vvssn vssn 0 0  * IO ground
vvccpd vccpd 0 vpd  * Pre-drive supply voltage

* Instantiate I/O Buffer
xhio_buf din oeb opdrain die rambh
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup vccn vccpd vcpad0 hio_buf
* Internal Loading on Pad
  * - This pad has an LVDS input buffer connected to it, along
  * with differential OCT circuitry. Both are disabled but
  * introduce loading on the pad that is modeled below.
  * xl_vs_input_load  die vss vccn lvds_input_load
  * xl_vs_oct_load  die vss vccpd vccn vcpad0 vccn lvds_oct_load

* I/O Buffer Package Model
  * - Single-ended I/O standard on a Row I/O
  * .lib 'lib/package.lib' hio
  * xpkg_die pin hio_pkg

Board and Trace Termination

The board trace and termination block of the simulation SPICE deck is provided only as an example. Replace this block with your specific board loading models.

Board Trace and Termination Block

* I/O Board Trace And Termination Description
  * - Replace this with your board trace and termination description
  * wline pin vssn load vssn N=1 L=1 RLGCMODEL=tlinemodel
  * .MODEL tlinemodel W MODELTYPE=RLGC N=1 Lo=7.13n Co=2.85p
  * Rterm2 load vssn 1x

Double-Counting Compensation Circuitry

The double-counting compensation circuitry block of the simulation SPICE deck instantiates a second I/O buffer that is used to measure double-counting. The buffer is configured identically to the user I/O buffer but is connected to the Quartus Prime software test load. The simulated delay of this second buffer can be interpreted as the amount of double-counting between the Quartus Prime software and HSPICE Writer simulated results.

As the amount of double-counting is constant for a given I/O standard on a given pin, consider separating the double-counting circuitry from the simulation file. In doing so, you can perform any number of I/O simulations while referencing the delay only once.

(Part of) Double-Counting Compensation Circuitry Block

* Double Counting Compensation Circuitry
  * The following circuit is designed to calculate the amount of
double counting between Quartus Prime and the HSPICE models. If
* you have not changed the default simulation temperature or
* transistor corner the double counting will be automatically
* compensated by this spice deck. In the event you wish to
* simulate an I/O at a different temperature or transistor corner
* you will need to remove this section of code and manually
* account for double counting. A description of Altera’s
* recommended procedure for this process can be found in the
* Quartus Prime HSPICE Writer AppNote.
  *
* Supply Voltages Settings
  .param vcn_tl=3.135
  .param vpd_tl=2.97

* Test Load Constant Definition
  opdrain_tl  0  0
  opdramb_tl  0  0
vrpullup_tl   rpullup_tl   0     0
* Instantiate Power Supplies
vvccn_tl      vccn_tl      0     vcn_tl
vsssn_tl      vssn_tl      0     0
vvccpd_tl     vccpd_tl     0     vpd_tl

* Instantiate I/O Buffer
xhio_testload din oeb opdrain tl die_tl rambh_tl
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup_tl vccn_tl vccpd_tl vcpad0_tl hio_buf

* Internal Loading on Pad
xlvds_input_testload die_tl vss vccn_tl lvds_input_load
xlvds_oct_testload die_tl vss vccpd_tl vccn_tl vcpad0_tl vccn_tl lvds_oct_load

* I/O Buffer Package Model
* - Single-ended I/O standard on a Row I/O
.lib 'lib/package.lib' hio
xpkg die pin hio_pkg

* Default Altera Test Load
* - 3.3V LVTTL default test condition is an open load

Related Information
The Double Counting Problem in HSPICE Simulations on page 14-17

Simulation Analysis
The simulation analysis block is set up to measure double-counting corrected delays. This is accomplished by measuring the uncompensated delay of the I/O buffer when connected to the user load, and when subtracting the simulated amount of double-counting from the test load I/O buffer.

Simulation Analysis Block
*Simulation Analysis Setup
* Print out the voltage waveform at both the pin and far end load
.print tran v(pin) v(load)
.tran 0.020ns 17ns
* Measure the propagation delay to the load pin. This value will include some double counting with Quartus Prime’s Tco
.measure TRAN tpd_uncomp_rise TRIG v(din) val='vc*0.5’ rise=1
+ TARG v(load) val='vcn*0.5’ rise=1
.measure TRAN tpd_uncomp_fall TRIG v(din) val='vc*0.5’ fall=1
+ TARG v(load) val='vcn*0.5’ fall=1

* The test load buffer can calculate the amount of double counting
.measure TRAN t_dblcnt_rise TRIG v(din) val='vc*0.5’ rise=1
+ TARG v(pin_tl) val='vcn_tl*0.5’ rise=1
.measure TRAN t_dblcnt_fall TRIG v(din) val='vc*0.5’ fall=1
+ TARG v(pin_tl) val='vcn_tl*0.5’ fall=1

* Calculate the true propagation delay by subtraction
.measure TRAN tpd_rise PARAM='tpd_uncomp_rise-t_dblcnt_rise'
.measure TRAN tpd_fall PARAM='tpd_uncomp_fall-t_dblcnt_fall'
Advanced Topics

The information in this section describes some of the more advanced topics and methods employed when setting up and running HSPICE simulation files.

PVT Simulations

The automatically generated HSPICE simulation files are set up to simulate the slow process corner using low voltage, high temperature, and slow transistors. To ensure a fully robust link, Altera recommends that you run simulations over all process corners.

To perform process, voltage, and temperature (PVT) simulations, manually modify the spice decks in a two step process:

1. Remove the double-counting compensation circuitry from the simulation file. This is required as the amount of double-counting is dependant upon how the Quartus Prime software calculates delays and is not based on which PVT corner is being simulated. By default, the Quartus Prime software provides timing numbers using the slow process corner.
2. Select the proper corner for the PVT simulation by setting the correct HSPICE temperature, changing the supply voltage sources, and loading the correct transistor models.

A more detailed description of HSPICE process corners can be found in the family-specific HSPICE model documentation.

Related Information

Accessing HSPICE Simulation Kits on page 14-16

Hold Time Analysis

Altera recommends performing worst-case hold time analysis using the fast corner models, which use fast transistors, high voltage, and low temperature. This involves modifying the SPICE decks to select the correct temperature option, change the supply voltage sources, and load the correct fast transistor models. The values of these parameters are located in the header comment section of the corresponding simulation deck files.

For a truly worst-case analysis, combine the HSPICE Writer hold time analysis results with the Quartus Prime software fast timing model. This requires that you change the double-counting compensation circuitry in the simulations files to also simulate the fast process corners, as this is what the Quartus Prime software uses for the fast timing model.

Note: This method of hold time analysis is recommended only for globally synchronous buses. Do not apply this method of hold-time analysis to source synchronous buses. This is because the source synchronous clocking scheme is designed to cancel out some of the PVT timing effects. If this is not taken into account, the timing results will not be accurate. Proper source synchronous timing analysis is beyond the scope of this document.

I/O Voltage Variations

Use each of the FPGA family datasheets to verify the recommended operating conditions for supply voltages. For current FPGA families, the maximum recommended voltage corresponds to the fast corner, while the minimum recommended voltage corresponds to the slow corner. These voltage recommendations are specified at the power pins of the FPGA and are not necessarily the same voltage that are seen by the I/O buffers due to package IR drops.

The automatically generated HSPICE simulation files model this IR effect pessimistically by including a 50-mV IR drop on the $V_{CCPD}$ supply when a high drive strength standard is being used.
Correlation Report
Correlation reports for the HSPICE I/O models are located in the family-specific HSPICE I/O buffer simulation kits.

Related Information
Accessing HSPICE Simulation Kits on page 14-16

Table 14-2: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Updated device support.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Was volume 3, chapter 12 in the 8.1.0 release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>• Changed to 8-1/2 x 11 page size.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added information for Stratix III devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Input signals for Cyclone III devices are supported.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| May 2008 | 8.0.0   | • Updated “Introduction” on page 12–1.  
• Updated Figure 12–1.  
• Updated Figure 12–3.  
• Updated Figure 12–13.  
• Updated “Output File Generation” on page 12–6.  
• Updated “Simulation with HSPICE Models” on page 12–17.  
• Updated “Invoking HSPICE Writer from the Command Line” on page 12–22.  
• Added “Sample Input for I/O HSPICE Simulation Deck” on page 12–29.  
• Added “Sample Output for I/O HSPICE Simulation Deck” on page 12–33.  
• Updated “Correlation Report” on page 12–41.  
• Added hyperlinks to referenced documents and websites throughout the chapter.  
• Made minor editorial updates. |

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
The Quartus Prime software interacts with the following software to provide a complete FPGA-to-board integration design workflow: the Cadence Allegro Design Entry HDL software and the Cadence Allegro Design Entry CIS (Component Information System) software (also known as OrCAD Capture CIS). The information is useful for board design and layout engineers who want to begin the FPGA board integration process while the FPGA is still in the design phase. Part librarians can also benefit by learning the method to use output from the Quartus Prime software to create new library parts and symbols.

With today’s large, high-pin-count and high-speed FPGA devices, good PCB design practices are important to ensure the correct operation of your system. The PCB design takes place concurrently with the design and programming of the FPGA. An FPGA or ASIC designer initially creates the signal and pin assignments and the board designer must transfer these assignments to the symbols used in their system circuit schematics and board layout correctly. As the board design progresses, you must perform pin reassignments to optimize the layout. You must communicate pin reassignments to the FPGA designer to ensure the new assignments are processed through the FPGA with updated placement and routing.

You require the following software:

- The Quartus Prime software version 5.1 or later
- The Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software version 15.2 or later
- The OrCAD Capture software with the optional CIS option version 10.3 or later (optional)

**Note:** These programs are very similar because the Cadence Allegro Design Entry CIS software is based on the OrCAD Capture software. Any procedural information can also apply to the OrCAD Capture software unless otherwise noted.

**Related Information**

- [www.cadence.com](http://www.cadence.com)  
  For more information about obtaining and licensing the Cadence tools and for product information, support, and training
- [www.cadence.com](http://www.cadence.com)  
  For more information about the OrCAD Capture software and the CIS option
- [www.ema-eda.com](http://www.ema-eda.com)  
  For more information about Cadence and OrCAD support and training
Product Comparison

Table 15-1: Cadence and OrCAD Product Comparison

<table>
<thead>
<tr>
<th>Description</th>
<th>Cadence Allegro Design Entry HDL</th>
<th>Cadence Allegro Design Entry CIS</th>
<th>OrCAD Capture CIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Former Name</td>
<td>Concept HDL Expert</td>
<td>Capture CIS Studio</td>
<td>—</td>
</tr>
<tr>
<td>History</td>
<td>More commonly known by its former name, Cadence renamed all board design tools in 2004 under the Allegro name.</td>
<td>Based directly on OrCAD Capture CIS, the Cadence Allegro Design Entry CIS software is still developed by OrCAD but sold and marketed by Cadence. EMA provides support and training.</td>
<td>The basis for Design Entry CIS is still developed by OrCAD for continued use by existing OrCAD customers. EMA provides support and training for all OrCAD products.</td>
</tr>
<tr>
<td>Vendor Design Flow</td>
<td>Cadence Allegro 600 series, formerly known as the Expert Series, for high-end, high-speed design.</td>
<td>Cadence Allegro 200 series, formerly known as the Studio Series, for small- to medium-level design.</td>
<td>—</td>
</tr>
</tbody>
</table>

Related Information

- www.cadence.com
- www.ema-eda.com

FPGA-to-PCB Design Flow

You can create a design flow integrating an Altera FPGA design from the Quartus Prime software through a circuit schematic in the Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software.
To create FPGA symbols using the Cadence Allegro PCB Librarian Part Developer tool, you must obtain the Cadence PCB Librarian Expert license. You can update symbols with changes made to the FPGA design using any of these tools.
Integrating Altera FPGA Design

To integrate an Altera FPGA design starting in the Quartus Prime software through to a circuit schematic in the Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software, follow these steps:

1. In the Quartus Prime software, compile your design to generate a Pin-Out File (.pin) to transfer the assignments to the Cadence software.

2. If you are using the Cadence Allegro Design Entry HDL software for your schematic design, follow these steps:
   a. Open an existing project or create a new project in the Cadence Allegro Project Manager tool.
   b. Construct a new symbol or update an existing symbol using the Cadence Allegro PCB Librarian Part Developer tool.
   c. With the Cadence Allegro PCB Librarian Part Developer tool, edit your symbol or fracture it into smaller parts (optional).
   d. Instantiate the symbol in your Cadence Allegro Design Entry HDL software schematic and transfer the design to your board layout tool.

   or

   If you are using the Cadence Allegro Design Entry CIS software for your schematic design, follow these steps:
   e. Generate a new part in a new or existing Cadence Allegro Design Entry CIS project, referencing the .pin output file from the Quartus Prime software. You can also update an existing symbol with a new .pin.
   f. Split the symbol into smaller parts as necessary.
   g. Instantiate the symbol in your Cadence Allegro Design Entry CIS schematic and transfer the design to your board layout tool.

Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA

With the Quartus Prime software, you can extract pin assignment data and perform SSN analysis of your FPGA design for designs targeting the Stratix III device family.

You can analyze SSN in your device early in the board layout stage as part of your overall pin planning process; however, you do not have to perform SSN analysis to generate pin assignment data from the Quartus Prime software. You can use the SSN Analyzer tool to optimize the pin assignments for better SSN performance of your device.

Related Information

- Simultaneous Switching Noise (SSN) Analysis and Optimizations

Setting Up the Quartus Prime Software

You can transfer pin and signal assignments from the Quartus Prime software to the Cadence design tools by generating the Quartus Prime project .pin. The .pin is an output file generated by the Quartus Prime Fitter containing pin assignment information. You can use the Quartus Prime Pin Planner to set and change the assignments in the .pin and then transfer the assignments to the Cadence design tools.
cannot, however, import pin assignment changes from the Cadence design tools into the Quartus Prime software with the .pin.

The .pin lists all used and unused pins on your selected Altera device. The .pin also provides the following basic information fields for each assigned pin on the device:

- Pin signal name and usage
- Pin number
- Signal direction
- I/O standard
- Voltage
- I/O bank
- User or Fitter-assigned

Related Information

- I/O Management on page 2-1
  For more information about using the Quartus Prime Pin Planner to create or change pin assignment details.

Generating a .pin File

To generate a .pin, follow these steps:

1. Compile your design.
2. Locate the .pin in your Quartus Prime project directory with the name <project name>.pin.

Related Information

- I/O Management on page 2-1
  For more information about pin and signal assignment transfer and the files that the Quartus Prime software can import and export.

FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software

The Cadence Allegro Design Entry HDL software is a schematic capture tool and is part of the Cadence 600 series design flow. Use the Cadence Allegro Design Entry HDL software to create flat circuit schematics for all types of PCB design. The Cadence Allegro Design Entry HDL software can also create hierarchical schematics to facilitate design reuse and team-based design. With the Cadence Allegro Design Entry HDL software, the design flow from FPGA-to-board is one-way, using only the .pin generated by the Quartus Prime software. You can only make signal and pin assignment changes in the Quartus Prime software and these changes reflect as updated symbols in a Cadence Allegro Design Entry HDL project.

For more information about the design flow with the Cadence Allegro Design Entry HDL software, refer to Figure 15-1.

Note: Routing or pin assignment changes made in a board layout tool or a Cadence Allegro Design Entry HDL software symbol cannot be back-annotated to the Quartus Prime software.
Creating Symbols

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required when designing a new PCB. Schematics must understand how the PCB works, and to generate a netlist for a board layout tool for board design and routing. The Cadence Allegro PCB Librarian Part Developer tool allows you to create schematic symbols based on FPGA designs exported from the Quartus Prime software.

You can create symbols for the Cadence Allegro Design Entry HDL project with the Cadence Allegro PCB Librarian Part Developer tool, which is available in the Cadence Allegro Project Manager tool. Altera recommends using the Cadence Allegro PCB Librarian Part Developer tool to import FPGA designs into the Cadence Allegro Design Entry HDL software.

You must obtain a PCB Librarian Expert license from Cadence to run the Cadence Allegro PCB Librarian Part Developer tool. The Cadence Allegro PCB Librarian Part Developer tool provides a GUI with many options for creating, editing, fracturing, and updating symbols. If you do not use the Cadence Allegro PCB Librarian Part Developer tool, you must create and edit symbols manually in the Symbol Schematic View in the Cadence Allegro Design Entry HDL software.

Note: If you do not have a PCB Librarian Expert license, you can automatically create FPGA symbols using the programmable IC (PIC) design flow found in the Cadence Allegro Project Manager tool.

Before creating a symbol from an FPGA design, you must open a Cadence Allegro Design Entry HDL project with the Cadence Allegro Project Manager tool. If you do not have an existing Cadence Allegro Design Entry HDL project, you can create one with the Cadence Allegro Design Entry HDL software. The Cadence Allegro Design Entry HDL project directory with the name <project name>.cpm contains your Cadence Allegro Design Entry HDL projects.

While the Cadence Allegro PCB Librarian Part Developer tool refers to symbol fractures as slots, the other tools use different names to refer to symbol fractures.

Table 15-2: Symbol Fracture Naming Conventions

<table>
<thead>
<tr>
<th></th>
<th>Cadence Allegro PCB Librarian Part Developer Tool</th>
<th>Cadence Allegro Design Entry HDL Software</th>
<th>Cadence Allegro Design Entry CIS Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>During symbol generation</td>
<td>Slots</td>
<td>—</td>
<td>Sections</td>
</tr>
<tr>
<td>During symbol schematic instantiation</td>
<td>—</td>
<td>Versions</td>
<td>Parts</td>
</tr>
</tbody>
</table>

Related Information

www.cadence.com
Provides information about using the PIC design flow.
You can create, fracture, and edit schematic symbols for your designs using the Cadence Allegro PCB Librarian Part Developer tool. Symbols designed in the Cadence Allegro PCB Librarian Part Developer tool can be split or fractured into several functional blocks called slots, allowing multiple smaller part fractures to exist on the same schematic page or across multiple pages.

Cadence Allegro PCB Librarian Part Developer Tool in the Design Flow

To run the Cadence Allegro PCB Librarian Part Developer tool, you must open a Cadence Allegro Design Entry HDL project in the Cadence Allegro Project Manager tool. To open the Cadence Allegro PCB Librarian Part Developer tool, on the Flows menu, click **Library Management**, and then click **Part Developer**.

**Related Information**

- **FPGA-to-PCB Design Flow** on page 15-2

**Import and Export Wizard**

After starting the Cadence Allegro PCB Librarian Part Developer tool, use the **Import and Export** wizard to import your pin assignments from the Quartus Prime software.

**Note:** Altera recommends using your PCB Librarian Expert license file. To point to your PCB Librarian Expert license file, on the File menu, click **Change Product** and then select the correct product license.

To access the Import and Export wizard, follow these steps:

1. On the File menu, click **Import and Export**.
2. Select **Import ECO-FPGA**, and then click **Next**.
3. In the **Select Source** page of the **Import and Export** wizard, specify the following settings:
a. In the Vendor list, select Altera.
b. In the PnR Tool list, select quartusII.
c. In the PR File box, browse to select the .pin in your Quartus Prime project directory.
d. Click Simulation Options to select simulation input files.
e. Click Next.

4. In the Select Destination dialog box, specify the following settings:
   a. Under Select Component, click Generate Custom Component to create a new component in a library,
      or
      Click Use standard component to base your symbol on an existing component.

   **Note:** Altera recommends creating a new component if you previously created a generic component for an FPGA device. Generic components can cause some problems with your design. When you create a new component, you can place your pin and signal assignments from the Quartus Prime software on this component and reuse the component as a base when you have a new FPGA design.

   b. In the Library list, select an existing library. You can select from the cells in the selected library. Each cell represents all the symbol versions and part fractures for a particular part. In the Cell list, select the existing cell to use as a base for your part.
   c. In the Destination Library list, select a destination library for the component. Click Next.
   d. Review and edit the assignments you import into the Cadence Allegro PCB Librarian Part Developer tool based on the data in the .pin and then click Finish. The location of each pin is not included in the Preview of Import Data page of the Import and Export wizard, but input pins are on the left side of the created symbol, output pins on the right, power pins on the top, and ground pins on the bottom.

**Editing and Fracturing Symbol**

After creating your new symbol in the Cadence Allegro PCB Librarian Part Developer tool, you can edit the symbol graphics, fracture the symbol into multiple slots, and add or change package or symbol properties.

The Part Developer Symbol Editor contains many graphical tools to edit the graphics of a particular symbol. To edit the symbol graphics, select the symbol in the cell hierarchy. The Symbol Pins tab appears. You can edit the preview graphic of the symbol in the Symbol Pins tab.

Fracturing a Cadence Allegro PCB Librarian Part Developer package into separate symbol slots is useful for FPGA designs. A single symbol for most FPGA packages might be too large for a single schematic page. Splitting the part into separate slots allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you can create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.
To fracture a part into separate slots, or to modify the slot locations of pins on parts fractured in the Cadence Allegro PCB Librarian Part Developer tool, follow these steps:

1. Start the Cadence Allegro Design Project Manager.
3. Click Part Developer.
4. Click the name of the package you want to change in the cell hierarchy.
5. Click Functions/Slots. If you are not creating new slots but want to change the slot location of some pins, proceed to Step 6. If you are creating new slots, click Add. A dialog box appears, allowing you to add extra symbol slots. Set the number of extra slots you want to add to the existing symbol, not the total number of desired slots for the part. Click OK.
6. Click Distribute Pins. Specify the slot location for each pin. Use the checkboxes in each column to move pins from one slot to another. Click OK.
7. After distributing the pins, click the Package Pin tab and click Generate Symbol(s).
8. Select whether to create a new symbol or modify an existing symbol in each slot. Click OK.

The newly generated or modified slot symbols appear as separate symbols in the cell hierarchy. Each of these symbols can be edited individually.

**Caution:** The Cadence Allegro PCB Librarian Part Developer tool allows you to remap pin assignments in the Package Pin tab of the main Cadence Allegro PCB Librarian Part Developer window. If signals remap to different pins in the Cadence Allegro PCB Librarian Part Developer tool, the changes reflect only in regenerated symbols for use in your schematics. You cannot transfer pin assignment changes to the Quartus Prime software from the Cadence Allegro PCB Librarian Part Developer tool, which creates a potential mismatch of the schematic symbols and assignments in the FPGA design. If pin assignment changes are necessary, make the changes in the Quartus Prime Pin Planner instead of the Cadence Allegro PCB Librarian Part Developer tool, and update the symbol as described in the following sections.

For more information about creating, editing, and organizing component symbols with the Cadence Allegro PCB Librarian Part Developer tool, refer to the Part Developer Help.
Updating FPGA Symbols

As the design process continues, you must make logic changes in the Quartus Prime software, placing signals on different pins after recompiling the design, or use the Quartus Prime Pin Planner to make changes manually. The board designer can request such changes to improve the board routing and layout. To ensure signals connect to the correct pins on the FPGA, you must carry forward these types of changes to the circuit schematic and board layout tools. Updating the *.pin in the Quartus Prime software facilitates this flow.

Figure 15-4: Updating the FPGA Symbol in the Design Flow

To update the symbol using the Cadence Allegro PCB Librarian Part Developer tool after updating the *.pin, follow these steps:

1. On the File menu, click **Import and Export**. The Import and Export wizard appears.
2. In the list of actions to perform, select **Import ECO - FPGA**. Click **Next**. The Select Source dialog box appears.
3. Select the updated source of the FPGA assignment information. In the **Vendor** list, select **Altera**. In the **PnR Tool** list, select **quartusII**. In the **PR File** field, click **browse** to specify the updated *.pin in your Quartus Prime project directory. Click **Next**. The Select Destination window appears.
4. Select the source component and a destination cell for the updated symbol. To create a new component based on the updated pin assignment data, select **Generate Custom Component**. Selecting **Generate Custom Component** replaces the cell listed under the **Specify Library and Cell** name header with a new, nonfractured cell. You can preserve these edits by selecting **Use standard component and select the existing library and cell**. Select the destination library for the component and click **Next**. The **Preview of Import Data** dialog box appears.
5. Make any additional changes to your symbol. Click **Next**. A list of ECO messages appears summarizing the changes made to the cell. To accept the changes and update the cell, click **Finish**.
6. The main Cadence Allegro PCB Librarian Part Developer window appears. You can edit, fracture, and generate the updated symbols as usual from the main Cadence Allegro PCB Librarian Part Developer window.

**Note:** If the Cadence Allegro PCB Librarian Part Developer tool is not set up to point to your PCB Librarian Expert license file, an error message appears in red at the bottom of the message text.
window of the Part Developer when you select the **Import and Export** command. To point to your PCB Librarian Expert license, on the File menu, click **Change Product**, and select the correct product license.

**Related Information**
- **FPGA-to-PCB Design Flow** on page 15-2

### Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software

To instantiate the symbol in your Cadence Allegro Design Entry HDL schematic after saving the new symbol in the Cadence Allegro PCB Librarian Part Developer tool, follow these steps:

1. In the Cadence Allegro Project Manager tool, switch to the board design flow.
2. On the Flows menu, click **Board Design**.
3. To start the Cadence Allegro Design Entry HDL software, click **Design Entry**.
4. To add the newly created symbol to your schematic, on the Component menu, click **Add**. The **Add Component** dialog box appears.
5. Select the new symbol library location, and select the name of the cell you created from the list of cells.

The symbol attaches to your cursor for placement in the schematic. To fracture the symbol into slots, right-click the symbol and choose **Version** to select one of the slots for placement in the schematic.

**Related Information**
- [www.cadence.com](http://www.cadence.com)
  Provides more information about the Cadence Allegro Design Entry HDL software, including licensing, support, usage, training, and product updates.

### FPGA-to-Board Integration with Cadence Allegro Design Entry CIS Software

The Cadence Allegro Design Entry CIS software is a schematic capture tool (part of the Cadence 200 series design flow based on OrCAD Capture CIS). Use the Cadence Allegro Design Entry CIS software to create flat circuit schematics for all types of PCB design. You can also create hierarchical schematics to facilitate design reuse and team-based design using the Cadence Allegro Design Entry CIS software. With the Cadence Allegro Design Entry CIS software, the design flow from FPGA-to-board is unidirectional using only the `.pin` generated by the Quartus Prime software. You can only make signal and pin assignment changes in the Quartus Prime software. These changes reflect as updated symbols in a Cadence Allegro Design Entry CIS schematic project.
Creating a Cadence Allegro Design Entry CIS Project

The Cadence Allegro Design Entry CIS software has built-in support for creating schematic symbols using pin assignment information imported from the Quartus Prime software.

To create a new project in the Cadence Allegro Design Entry CIS software, follow these steps:

1. On the File menu, point to New and click Project. The New Project wizard starts.

   When you create a new project, you can select the PC Board wizard, the Programmable Logic wizard, or a blank schematic.

2. Select the PC Board wizard to create a project where you can select which part libraries to use, or select a blank schematic.

The Programmable Logic wizard only builds an FPGA logic design in the Cadence Allegro Design Entry CIS software.

Note: Routing or pin assignment changes made in a board layout tool or a Cadence Allegro Design Entry CIS symbol cannot be back-annotated to the Quartus Prime software.

Related Information

- www.cadence.com
  For more information about the Cadence Allegro Design Entry CIS software, including licensing, support, usage, training, and product updates.

- www.ema-eda.com
  For more information about the Cadence Allegro Design Entry CIS software, including licensing, support, usage, training, and product updates.
Your new project is in the specified location and consists of the following files:

- OrCAD Capture Project File (.opj)
- Schematic Design File (.dsn)

### Generating a Part

After you create a new project or open an existing project in the Cadence Allegro Design Entry CIS software, you can generate a new schematic symbol based on your Quartus Prime FPGA design. You can also update an existing symbol. The Cadence Allegro Design Entry CIS software stores component symbols in OrCAD Library File (.olb). When you place a symbol in a library attached to a project, it is immediately available for instantiation in the project schematic.

You can add symbols to an existing library or you can create a new library specifically for the symbols generated from your FPGA designs. To create a new library, follow these steps:

1. On the File menu, point to New and click Library in the Cadence Allegro Design Entry CIS software to create a default library named library1.olb. This library appears in the Library folder in the Project Manager window of the Cadence Allegro Design Entry CIS software.
2. To specify a desired name and location for the library, right-click the new library and select Save As. Saving the new library creates the library file.

### Generating Schematic Symbol

You can now create a new symbol to represent your FPGA design in your schematic.

To generate a schematic symbol, follow these steps:

1. Start the Cadence Allegro Design Entry CIS software.
2. On the Tools menu, click Generate Part. The Generate Part dialog box appears.
3. To specify the .pin from your Quartus Prime design, in the Netlist/source file type field, click Browse.
4. In the Netlist/source file type list, select Altera Pin File
5. Type the new part name.
6. Specify the Destination part library for the symbol. Failing to select an existing library for the part creates a new library with a default name that matches the name of your Cadence Allegro Design Entry CIS project.
7. To create a new symbol for this design, select Create new part. If you updated your .pin in the Quartus Prime software and want to transfer any assignment changes to an existing symbol, select Update pins on existing part in library.
8. Select any other desired options and set Implementation type to <none>. The symbol is for a primitive library part based only on the .pin and does not require special implementation. Click OK.
9. Review the Undo warning and click Yes to complete the symbol generation.

You can locate the generated symbol in the selected library or in a new library found in the Outputs folder of the design in the Project Manager window. Double-click the name of the new symbol to see its graphical representation and edit it manually using the tools available in the Cadence Allegro Design Entry CIS software.

**Note:** For more information about creating and editing symbols in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.
Splitting a Part

After saving a new symbol in a project library, you can fracture the symbol into multiple parts called sections. Fracturing a part into separate sections is useful for FPGA designs. A single symbol for most FPGA packages might be too large for a single schematic page. Splitting the part into separate sections allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you can create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.

**Figure 15-6: Splitting a Symbol into Multiple Sections**

![Diagram showing symbol sections](image)

* This diagram represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes might have different sets of configuration pins, but can be fractured in the same manner.

* The power/ground section shows only a representation of power and ground pins because the device contains a high number of power and ground pins.

**Note:** Although symbol generation in the Design Entry CIS software refers to symbol fractures as sections, other tools use different names to refer to symbol fractures.

To split a part into sections, select the part in its library in the Project Manager window of the Cadence Allegro Design Entry CIS software. On the Tools menu, click **Split Part** or right-click the part and choose **Split Part**. The **Split Part Section Input Spreadsheet** appears.
Each row in the spreadsheet represents a pin in the symbol. The **Section** column indicates the section of the symbol to which each pin is assigned. You can locate all pins in a new symbol in section 1. You can change the values in the **Section** column to assign pins to various sections of the symbol. You can also specify the side of a section on the location of the pin by changing the values in the **Location** column. When you are ready, click **Split**. A new symbol appears in the same library as the original with the name `<original part name>_Split1`.

View and edit each section individually. To view the new sections of the part, double-click the part. The Part Symbol Editor window appears and the first section of the part displays for editing. On the View menu, click **Package** to view thumbnails of all the part sections. To edit the section of the symbol, double-click the thumbnail.

For more information about splitting parts into sections and editing symbol sections in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

**Instantiating a Symbol in a Design Entry CIS Schematic**

After saving a new symbol in a library in your Cadence Allegro Design Entry CIS project, you can instantiate the new symbol on a page in your schematic. Open a schematic page in the Project Manager window of the Cadence Allegro Design Entry CIS software. To add the newly created symbol to your schematic on the schematic page, on the Place menu, click **Part**. The **Place Part** dialog box appears.
Select the new symbol library location and the newly created part name. If you select a part that is split into sections, you can select the section to place from the Part pop-up menu. Click OK. The symbol attaches to your cursor for placement in the schematic. To place the symbol, click on the schematic page.

For more information about using the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Altera Libraries for the Cadence Allegro Design Entry CIS Software

Altera provides downloadable .olb for many of its device packages. You can add these libraries to your Cadence Allegro Design Entry CIS project and update the symbols with the pin assignments contained in the .pin generated by the Quartus Prime software. You can use the downloaded library symbols as a base for creating custom schematic symbols with your pin assignments that you can edit or fracture. This method increases productivity by reducing the amount of time it takes to create and edit a new symbol.

Using the Altera-provided Libraries with your Cadence Allegro Design Entry CIS Project

To use the Altera-provided libraries with your Cadence Allegro Design Entry CIS project, follow these steps:

1. Download the library of your target device from the Download Center page found through the Support page on the Altera website.
2. Create a copy of the appropriate .olb to maintain the original symbols. Place the copy in a convenient location, such as your Cadence Allegro Design Entry CIS project directory.
3. In the Project Manager window of the Cadence Allegro Design Entry CIS software, click once on the Library folder to select it. On the Edit menu, click Project or right-click the Library folder and choose Add File to select the copy of the downloaded .olb and add it to your project. You can locate the new library in the list of part libraries for your project.
5. In the Netlist/source file field, click Browse to specify the .pin in your Quartus Prime design.
6. From the Netlist/source file type list, select Altera Pin File.
7. For Part name, type the name of the target device the same as it appears in the downloaded library file. For example, if you are using a device from the CYCLONE06.OLB library, type the part name to match one of the devices in this library such as ep1c6f256. You can rename the symbol in the Project Manager window after updating the part.
8. Set the **Destination part library** to the copy of the downloaded library you added to the project.

9. Select **Update pins on existing part in library**. Click **OK**.

10. Click **Yes**.

The symbol is updated with your pin assignments. Double-click the symbol in the Project Manager window to view and edit the symbol. On the View menu, click **Package** if you want to view and edit other sections of the symbol. If the symbol in the downloaded library is fractured into sections, you can edit each section but you cannot further fracture the part. You can generate a new part without using the downloaded part library if you require additional sections.

For more information about creating, editing, and fracturing symbols in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

### Document Revision History

**Table 15-3: Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Converted to DITA format.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• General style editing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Referenced Document Section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added a link to Help in “Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA” on page 9–5.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Added “Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA” on page 9–5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• General style editing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Edited Figure 9–4 on page 9–10 and Figure 9–8 on page 9–16.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Chapter 9 was previously Chapter 7 in the 8.1 software release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Updated references.</td>
</tr>
</tbody>
</table>
Related Information

Quartus Handbook Archive
For previous versions of the Quartus Prime Handbook, refer to the Quartus Handbook Archive.
You can integrate the Mentor Graphics® I/O Designer or DxDesigner PCB design tools into the Quartus Prime design flow. This combination provides a complete FPGA-to-board design workflow.

With today's large, high-pin-count and high-speed FPGA devices, good and correct PCB design practices are essential to ensure correct system operation. The PCB design takes place concurrently with the design and programming of the FPGA. The FPGA or ASIC designer initially creates signal and pin assignments, and the board designer must correctly transfer these assignments to the symbols in their system circuit schematics and board layout. As the board design progresses, Altera recommends reassigning pins to optimize the PCB layout. Ensure that you inform the FPGA designer of the pin reassignments so that the new assignments are included in an updated placement and routing of the design.

The Mentor Graphics I/O Designer software allows you to take advantage of the full FPGA symbol design, creation, editing, and back-annotation flow supported by the Mentor Graphics tools.

This chapter covers the following topics:

- Mentor Graphics and Altera software integration flow
- Generating supporting files
- Adding Quartus Prime I/O assignments to I/O Designer
- Updating assignment changes between the I/O Designer and the Quartus Prime software
- Generating I/O Designer symbols
- Creating DxDesigner symbols from the Quartus Prime output files

This chapter is intended for board design and layout engineers who want to start the FPGA board integration while the FPGA is still in the design phase. Alternatively, the board designer can plan the FPGA pin-out and routing requirements in the Mentor Graphics tools and pass the information back to the Quartus Prime software for placement and routing. Part librarians can also benefit from this chapter by learning how to use output from the Quartus Prime software to create new library parts and symbols.

The procedures in this chapter require the following software:

- The Quartus Prime software version 5.1 or later
- DxDesigner software version 2004 or later
- Mentor Graphics I/O Designer software (optional)

**Note:** To obtain and license the Mentor Graphics tools and for product information, support, and training, refer to the Mentor Graphics website.
FPGA-to-PCB Design Flow

You can create a design flow integrating an Altera® FPGA design from the Quartus Prime software, and a circuit schematic in the DxDesigner software.

**Figure 16-1: Design Flow with and Without the I/O Designer Software**
Note: The Quartus Prime software generates the .fx in the output directory you specify in the Board-Level page of the Settings dialog box. However, the Quartus Prime software and the I/O Designer software can import pin assignments from an .fx located in any directory. Use a backup .fx to prevent overwriting existing assignments or importing invalid assignments.

To integrate the I/O Designer into your design flow, follow these steps:

1. In the Quartus Prime software, click Assignments > Settings > EDA Tool Settings > Board-Level to specify settings for .fx symbol file generation.
2. Compile your design to generate the .fx and Pin-Out File (.pin) in the Quartus Prime project directory.
3. Create a board design with the DxDesigner software and the I/O Designer software by performing the following steps:
   a. Create a new I/O Designer database based on the .fx and the .pin files.
   b. In the I/O Designer software, make adjustments to signal and pin assignments.
   c. Regenerate the .fx in the I/O Designer software to export the I/O Designer software changes to the Quartus Prime software.
   d. Generate a single or fractured symbol for use in the DxDesigner software.
   e. Add the symbol to the sym directory of a DxDesigner project, or specify a new DxDesigner project with the new symbol.
   f. Instantiate the symbol in your DxDesigner schematic and export the design to the board layout tool.
   g. Back-annotate pin changes created in the board layout tool to the DxDesigner software and back to the I/O Designer software and the Quartus Prime software.
4. Create a board design with the DxDesigner software without the I/O Designer software by performing the following steps:
   a. Create a new DxBoardLink symbol with the Symbol wizard and reference the .pin from the Quartus Prime software in an existing DxDesigner project.
   b. Instantiate the symbol in your DxDesigner schematic and export the design to a board layout tool.

Note: You can update these symbols with design changes with or without the I/O Designer software. If you use the Mentor Graphics I/O Designer software and you change symbols with the DxDesigner software, you must reimport the symbols into I/O Designer to avoid overwriting your symbol changes.

## Integrating with I/O Designer

You can integrate the Mentor Graphics I/O Designer software into the Quartus Prime design flow. Pin and signal assignment changes can be made anywhere in the design flow with either the Quartus Prime Pin Planner or the I/O Designer software. The I/O Designer software facilitates moving these changes, as well as synthesis, placement, and routing changes, between the Quartus Prime software, an external synthesis tool (if used), and a schematic capture tool such as the DxDesigner software.

This section describes how to use the I/O Designer software to transfer pin and signal assignment information to and from the Quartus Prime software with an .fx, and how to create symbols for the DxDesigner software.
Note: (2) DxDesigner software-specific steps in the design flow are not part of the I/O Designer flow.

Generating Pin Assignment Files

You transfer I/O pin assignments from the Quartus Prime software to the Mentor Graphics PCB tools by generating optional .pin and .fx files during Quartus Prime compilation. These files contain pin assignment information for use in other tools. Click Assignments > Settings > Board-Level to specify settings for optional PCB tool file generation. Click Processing > Start Compilation to compile the design to generate the file(s) in the project directory.

The Quartus Prime-generated .pin contains the I/O pin name, number, location, direction, and I/O standard for all used and unused pins in the design. Click Assignments > Pin Planner to modify I/O pin
assignments. You cannot import pin assignment changes from a Mentor Graphics .pin into the Quartus Prime software.

The .fx is an input or output of either the Quartus Prime or I/O Designer software. You can generate an .fx in the Quartus Prime software for symbol generation in the Mentor Graphics I/O Designer software. A Quartus Prime .fx contains the pin name, number, location, direction, I/O standard, drive strength, termination, slew rate, IOB delay, and differential pins. An I/O Designer .fx additionally includes information about unused pins and pin set groups.

The I/O Designer software can also read from or update a Quartus Prime Settings File (.qsf). You can use the .qsf in the same way as use of the .fx, but pin swap group information does not transfer between I/O Designer and the Quartus Prime software. Use the .fx rather than the .qsf for transferring I/O assignment information.

Figure 16-3: Generating .pin and .fx files

**I/O Designer Settings**

You can directly export I/O Designer symbols to the DxDesigner software. To set options for integrating I/O Designer with Dx Designer, follow these steps:

1. Start the I/O Designer software.
2. Click Tools > Preferences.
3. Click Paths, and then double-click the DxDesigner executable file path field to select the location of the DxDesigner application.
4. Click Apply.
5. Click Symbol Editor, and then click Export. In the Export type menu, under General, select DxDesigner/PADS-Designer.
6. Click Apply, and then click OK.
Click File > Properties.

8. Click the PCB Flow tab, and then click Path to a DxDesigner project directory.

9. Click OK.

If you do not have a new DxDesigner project in the Database wizard and a DxDesigner project, you must create a new database with the DxDesigner software, and then specify the project location in I/O Designer.

Transferring I/O Assignments

You can transfer Quartus Prime signal and pin assignments contained in .pin and .fx files into an I/O Designer database. Use the I/O Designer Database Wizard to create a new database incorporating the .fx and .pin files. You can also create a new, empty database and manually add the assignment information. If there is no available signal or pin assignment information, you can create an empty database containing only a selection of the target device. This technique is useful if you know the signals in your design and the pins you want to assign. You can subsequently transfer this information to the Quartus Prime software for placement and routing.

Before you begin

You may create a very simple I/O Designer database that includes only the .pin or .fx file information. However, when using only a .pin, you cannot import I/O assignment changes from I/O Designer back into the Quartus Prime software without also generating an .fx. If your I/O Designer database includes only .fx file information, the database may not contain all the available I/O assignment information. The Quartus Prime .fx file only lists assigned pins. The .pin lists all assigned and unassigned device pins. Use both the .pin and the .fx to produce the most complete set of I/O Designer database information.

To create a new I/O Designer database using the Database wizard, follow these steps:

1. Start the I/O Designer software. The Welcome to I/O Designer dialog box appears. Select Wizard to create new database and click OK.

   If the Welcome to I/O Designer dialog box does not appear, you can access the wizard through the menu. To access the wizard, click File > Database Wizard.

2. Click Next. The Define HDL source file page appears.

   If no HDL files are available, or if the .fx contains your signal and pin assignments, you can skip Step 3 and proceed to Step 4.

3. If your design includes a Verilog HDL or VHDL file, you can add a top-level Verilog HDL or VHDL file in the I/O Designer software. Adding a file allows you to create functional blocks or get signal names from your design. You must create all physical pin assignments in I/O Designer if you are not using an .fx or a .pin. Click Next. The Database Name page appears.

4. In the Database Name page, type your database file name. Click Next. The Database Location window appears.

5. Add a path to the new or an existing database in the Location field, or browse to a database location. Click Next. The FPGA flow page appears.

6. In the Vendor menu, click Altera.

7. In the Tool/Library menu, click Quartus Prime <version> to select your version of the Quartus Prime software.

   Note: The Quartus Prime software version listed may not match your actual software version. If your version is not listed, select the latest version. If your target device is not available, the device may not yet be supported by the I/O Designer software.
8. Select the appropriate device family, device, package, and speed (if applicable), from the corresponding menus. Click Next. The Place and route page appears.

9. In the FPGAX file name field, type or browse to the backup copy of the .fx generated by the Quartus Prime software.

10. In the Pin report file name field, type or browse to the .pin generated by the Quartus Prime software. Click Next.

You can also select a .qsf for update. The I/O Designer software can update the pin assignment information in the .qsf without affecting any other information in the file.

Note: You can import a .pin without importing an .fx. The I/O Designer software does not generate a .pin. To transfer assignment information to the Quartus Prime software, select an additional file and file type. Altera recommends selecting an .fx in addition to a .pin for transferring all the assignment information in the .fx and .pin files. In some versions of the I/O Designer software, the standard file picker may incorrectly look for a .pin instead of an .fx. In this case, select All Files (*.*) from the Save as type list and select the file from the list.

11. On the Synthesis page, specify an external synthesis tool and a synthesis constraints file for use with the tool. If you do not use an external synthesis tool, click Next.

12. On the PCB Flow page, you can select an existing schematic project or create a new project as a symbol information destination.

   • To select an existing project, select Choose existing project and click Browse after the Project Path field. The Select project dialog box appears. Select the project.
   • To create a new project, in the Select project dialog box, select Create new empty project. Type the project file name in the Name field and browse to the location where you want to save the file. Click OK.

13. If you have not specified a design tool to which you can send symbol information in the I/O Designer software, click Advanced in the PCB Flow page and select your design tool. If you select the DxDesigner software, you have the option to specify a Hierarchical Occurrence Attributes (.oat) file to import into the I/O Designer software. Click Next and then click Finish to create the database.

Updating I/O Designer with Quartus Prime Pin Assignments

As you fine tune your design in the Quartus Prime software, changes to design logic and pin assignments are likely. You must transfer any pin assignment changes made during design iterations for correct analysis in your circuit schematic and board layout tools. You transfer Quartus Prime pin assignment changes to I/O Designer by updating the .fx and the .pin files in the Quartus Prime software. When you update the .fx or the .pin, the I/O Designer database imports the changes automatically when configured according to the following instructions.
Before you begin

Figure 16-4: Updating Quartus Prime Pin Assignments in I/O Designer

To update the .fx in your selected output directory and the .pin in your project directory after making changes to the design, perform the following tasks:

1. In the I/O Designer software, click File > Properties.
2. Under FPGA Xchange, specify the .fx file name and location.
3. Under Place and Route, specify the .pin file name and location.

   After you have set up these file locations, the I/O Designer software monitors these files for changes. If the specified .fx or .pin is modified during design processing, three indicators flash red in the lower right corner of the I/O Designer GUI. You can click the indicators to open the I/O Designer Update Wizard dialog box. The I/O Designer Update Wizard dialog box lists the updated files in the database.

4. Make logic or pin assignment changes in your design.
5. To preserve your changes an update the corresponding the .fx and .pin files, click Processing > Start > Start EDA Netlist Writer or Processing > Start Compilation.

   Note: Your I/O Designer database should us a backup copy of the .fx generated by the Quartus Prime software. Otherwise, updating the file in the Quartus Prime software overwrites any changes made to the file by the I/O Designer software. If there are I/O Designer assignments in the .fx that you want to preserve, create a backup copy of the file before updating it in the Quartus Prime software, and verify that your I/O Designer database points to the backup copy.

Updating Quartus Prime with I/O Designer Pin Assignments

As you fine tune your board design in I/O Designer, changes to signal routing and layout are likely. You must import any routing and layout changes into the Quartus Prime software for accurate place and route to match the new pin-out. The I/O Designer tool supports this flow.
Before you begin

Figure 16-5: Importing I/O Designer Pin Assignments

To import I/O Designer pin assignments, follow these steps:

1. Make pin assignment changes directly in the I/O Designer software, or the software can automatically update changes made in a board layout tool that are back-annotated to a schematic entry program such as the DxDesigner software.
2. To update the .fx with the changes, click **Generate > FPGA Xchange File**.
3. Open your Quartus Prime project.
4. Click **Assignments > Import Assignments**.
5. (Optional) To preserve original assignments before import, turn on **Copy existing assignments into <project name>.qsf.bak** before importing before importing the .fx.
6. Select the .fx and click **Open**.
7. Click **OK**.

Generating Schematic Symbols in I/O Designer

Circuit board schematic creation is one of the first tasks required in the design of a new PCB. You can use the I/O Designer software to generate schematic symbols for your Quartus Prime FPGA design for use in the DXDesigner schematic entry tools. The I/O Designer software can generate symbols for use in various Mentor Graphics schematic entry tools, and can import changes back-annotated by board layout tools to update the database and update the Quartus Prime software with the .fx.

Most FPGA devices contain hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. Symbol designs in the I/O Designer software can be split or fractured into various functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures join together with the use of the **hetero** attribute.
You can use the I/O Designer Symbol wizard to quickly create symbols that you can subsequently refine. Alternatively, you can import symbols from another DXDesigner project, and then assign an FPGA to the symbol. To import symbols in the I/O Designer software, File > Import Symbol.

I/O Designer symbols are either functional, physical (PCB), or both. Signals imported into the database, usually from Verilog HDL or VHDL files, are the basis of a functional symbol. No physical device pins must be associated with the signals to generate a functional symbol. This section focuses on board-level PCB symbols with signals directly mapped to physical device pins through assignments in either the Quartus Prime Pin Planner or in the I/O Designer database.

Generating Schematic Symbols
To create a symbol based on a selected Altera FPGA device, follow these steps:

1. Start the I/O Designer software.
2. Click Symbol > Symbol Wizard.
3. In the Symbol name field, type the symbol name. The DEVICE and PKG_TYPE fields display the device and package information.

   Note: If DEVICE and PKG_TYPE are blank or incorrect, close the Symbol wizard and specify the correct device information (File > Properties > FPGA Flow).

4. Under Symbol type, click PCB. Under Use signals, click All, then click Next.
5. Select fracturing options for your symbol. If you are using the Symbol wizard to edit a previously created fractured symbol, you must turn on Reuse existing fractures to preserve your current fractures. Select other options on this page as appropriate for your symbol. Click Next.
6. Select additional fracturing options for your symbol. Click Next.
7. Select the options for the appearance of the symbols. Click Next.
8. Define the information you want to label for the entire symbol and for individual pins. Click Next.
9. Add any additional signals and pins to the symbol. Click Finish.

   You can view your symbol and any fractures you created with the Symbol Editor. You can edit parts of the symbol, delete fractures, or rerun the Symbol wizard. When you modify pin assignments in I/O Designer database, I/O Designer symbols automatically reflect these changes. Modify assignments in the I/O Designer software by supplying and updated .fx from the Quartus Prime software, or by back-annotating changes in your board layout tool.

Exporting Schematic Symbols to DxDesigner
You can export your I/O Designer schematic symbols for to DxDesigner for further design entry work. To generate all fractures of a symbol, click Generate > All Symbols. To generate only the currently displayed symbol, click Generate > Current Symbol Only. The DxDesigner project /sym directory preserves each symbol in the database as a separate file. You can instantiate the symbols in your DxDesigner schematics.

Integrating with DxDesigner
You can integrate the Mentor Graphics DxDesigner schematic capture tool into the Quartus Prime design flow. Use DxDesigner to create flat circuit schematics or to create hierarchical schematics that facilitate design reuse and a team-based design for all PCB types. Use DxDesigner in conjunction with I/O Designer software for a complete FPGA I/O and PCB design flow.

If you use DxDesigner without the I/O Designer software, the design flow is one-way, using only the .pin generated by the Quartus Prime software. You can only make signal and pin assignment changes in the
Quartus Prime software. You cannot back-annotate changes made in a board layout tool or in a DxDesigner symbol to the Quartus Prime software.

**Figure 16-6: DxDesigner-only Flow (without I/O Designer)**

---

**DxDesigner Project Settings**

DxDesigner new projects automatically create FPGA symbols by default. However, if you are using the I/O Designer with DxDesigner, you must enable DxBoardLink Flow options for integration with the I/O Designer software. To enable the DxBoardLink flow design configuration when creating a new DxDesigner project, follow these steps:

1. Start the DxDesigner software.
2. Click **File > New**, and then click the **Project** tab.
3. Click **More**. Turn on **DxBoardLink**. To enable the DxBoardLink Flow design configuration for an existing project, click **Design Configurations** in the Design Configuration toolbar and turn on **DxBoardLink**.

**Creating Schematic Symbols in DxDesigner**

You can create schematic symbols in the DxDesigner software manually or with the Symbol wizard. The DxDesigner Symbol wizard is similar to the I/O Designer Symbol wizard, but with fewer fracturing options. The DxDesigner Symbol wizard creates, fractures, and edits FPGA symbols based on the specified Altera device. To create a symbol with the Symbol wizard, follow these steps:

1. Start the DxDesigner software.
2. Click **Symbol Wizard** in the toolbar.
3. Type the new symbol name in the name field and click **OK**.
4. Specify creation of a new symbol or modification of an existing symbol. To modify an existing symbol, specify the library path or alias, and select the existing symbol. To create a new symbol, select **DxBoardLink** for the symbol source. The DxDesigner block type defaults to Module because the FPGA design does not have an underlying DxDesigner schematic. Choose whether or not to fracture the symbol. Click **Next**.
5. Type a name for the symbol, an overall part name for all the symbol fractures, and a library name for the new library created for this symbol. By default, the part and library names are the same as the symbol name. Click **Next**.
6. Specify the appearance of the generated symbol and how it the grid you have set in your DxDesigner project schematic. After making your selections, Click Next.

7. In the FPGA vendor list, select Altera Quartus. In the Pin-Out file to import field, select the .pin from your Quartus Prime project directory. You can also specify Fracturing Scheme, Bus pin, and Power pin options. Click Next.

8. Select to create or modify symbol attributes for use in the DxDesigner software. Click Next.

9. On the Pin Settings page, make any final adjustments to pin and label location and information. Each tabbed spreadsheet represents a fracture of your symbol. Click Save Symbol.

After creating the symbol, you can examine and place any fracture of the symbol in your schematic. You can locate separate files of all the fractures you created in the library you specified or created in the /sym directory in your DxDesigner project. You can add the symbols to your schematics or you can manually edit the symbols or with the Symbol wizard.

Analyzing FPGA Simultaneous Switching Noise (SSN)

With the Quartus Prime software, you can extract pin assignment data and perform SSN analysis of your design. Perform SSN analysis early in the board layout stage as part of your overall pin planning process. Use the Quartus Prime SSN Analyzer to optimize the pin assignments for better SSN performance.

Scripting API

The I/O Designer software includes a command line Tcl interpreter. All commands input through the I/O Designer GUI translate into Tcl commands run by the tool. You can run individual Tcl commands or scripts in the I/O Designer Console window, rather than using the GUI.

You can use the following Tcl commands to control I/O Designer.

- set_fpga_xchange_file <file name>—specifies the .fx from which the I/O Designer software updates assignments.
- update_from_fpga_xchange_file—updates the I/O Designer database with assignment updates from the currently specified .fx.
- generate_fpga_xchange_file—updates the .fx with I/O Designer software changes for transfer back into the Quartus Prime software.
- symbolwizard—runs the I/O Designer Symbol wizard.
- set_dx_designer_project -path <path>

Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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. |
| June 2012    | 12.0.0  | • Removed survey link.                                                                                                                                 |
| December 2010| 10.1.0  | • Changed to new document template.                                                                                                                                |

**Related Information**

**Quartus Handbook Archive**

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

This document describes simulating designs that target Altera devices. Simulation verifies design behavior before device programming. The Quartus® Prime software supports RTL- and gate-level design simulation in supported EDA simulators. Simulation involves setting up your simulator working environment, compiling simulation model libraries, and running your simulation.

Simulator Support

The Quartus Prime software supports specific EDA simulator versions for RTL and gate-level simulation.

Table 1-1: Supported Simulators

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Simulator</th>
<th>Version</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aldec</td>
<td>Active-HDL</td>
<td>10.2 Update 2</td>
<td>Windows</td>
</tr>
<tr>
<td>Aldec</td>
<td>Riviera-PRO</td>
<td>2015.06</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Cadence</td>
<td>Incisive Enterprise</td>
<td>14.2</td>
<td>Linux</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td>ModelSim-Altera (provided)</td>
<td>10.4b</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td>ModelSim PE</td>
<td>10.4b</td>
<td>Windows</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td>ModelSim SE</td>
<td>10.4b</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td>QuestaSim</td>
<td>10.4b</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Synopsys</td>
<td>VCS/VCS MX</td>
<td>2014,12-SP1</td>
<td>Linux</td>
</tr>
</tbody>
</table>

Simulation Levels

The Quartus Prime software supports RTL and gate-level simulation of IP cores in supported EDA simulators.
<table>
<thead>
<tr>
<th>Simulation Level</th>
<th>Description</th>
<th>Simulation Input</th>
</tr>
</thead>
</table>
| RTL              | Cycle-accurate simulation using Verilog HDL, SystemVerilog, and VHDL design source code with simulation models provided by Altera and other IP providers. | • Design source/testbench  
• Altera simulation libraries  
• Altera IP plain text or IEEE encrypted RTL models  
• IP simulation models  
• Altera IP functional simulation models  
• Altera IP bus functional models  
• Qsys-generated models  
• Verification IP |
| Gate-level functional | Simulation using a post-synthesis or post-fit functional netlist testing the post-synthesis functional netlist, or post-fit functional netlist. | • Testbench  
• Altera simulation libraries  
• Post-synthesis or post-fit functional netlist  
• Altera IP bus functional models |
| Gate-level timing | Simulation using a post-fit timing netlist, testing functional and timing performance. Supported only for the Stratix IV, Cyclone IV, and MAX 10 device families. | • Testbench  
• Altera simulation libraries  
• Post-fit timing netlist  
• Post-fit Standard Delay Output File (.sdo). Not supported for MAX 10 devices. |

**Note:** Gate-level timing simulation of an entire design can be slow and should be avoided. Gate-level timing simulation is supported only for the Stratix IV and Cyclone IV device families. Use TimeQuest static timing analysis rather than gate-level timing simulation.
## HDL Support

The Quartus® Prime software provides the following HDL support for EDA simulators.

### Table 1-3: HDL Support

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
</table>
| **VHDL**         | • For VHDL RTL simulation, compile design files directly in your simulator. To use NativeLink automation, analyze and elaborate your design in the Quartus Prime software, and then use the NativeLink simulator scripts to compile the design files in your simulator. You must also compile simulation models from the Altera simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler or NativeLink to compile simulation models.  
  • For gate-level simulation, the EDA Netlist Writer generates a synthesized design netlist VHDL Output File (.vho). Compile the .vho in your simulator. You may also need to compile models from the Altera simulation libraries.  
  • IEEE 1364-2005 encrypted Verilog HDL simulation models are encrypted separately for each Altera-supported simulation vendor. If you want to simulate the model in a VHDL design, you need either a simulator that is capable of VHDL/Verilog HDL co-simulation, or any Mentor Graphics single language VHDL simulator. |
| **Verilog HDL**  | • For RTL simulation in Verilog HDL or SystemVerilog, compile your design files in your simulator. To use NativeLink automation, analyze and elaborate your design in the Quartus Prime software, and then use the NativeLink simulator scripts to compile your design files in your simulator. You must also compile simulation models from the Altera simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler or NativeLink to compile simulation models.  
  • For gate-level simulation, the EDA Netlist Writer generates a synthesized design netlist Verilog Output File (.vo). Compile the .vo in your simulator. |
| **SystemVerilog**|                                                                                                                                                                                                             |
| **Mixed HDL**    | • If your design is a mix of VHDL, Verilog HDL, and SystemVerilog files, you must use a mixed language simulator. Choose the most convenient supported language for generation of Altera IP cores in your design.  
  • Altera provides the entry-level ModelSim-Altera software, along with precompiled Altera simulation libraries, to simplify simulation of Altera designs. Starting in version 15.0, the ModelSim-Altera software supports native, mixed-language (VHDL/Verilog HDL/SystemVerilog) co-simulation of plain text HDL. If you have a VHDL-only simulator and need to simulate Verilog HDL modules and IP cores, you can either acquire a mixed-language simulator license from the simulator vendor, or use the ModelSim-Altera software. |
| **Schematic**    | You must convert schematics to HDL format before simulation. You can use the converted VHDL or Verilog HDL files for RTL simulation.                                                                                   |
## Simulation Flows

The Quartus® Prime software supports various methods for integrating your supported simulator into the design flow.

### Table 1-4: Simulation Flows

<table>
<thead>
<tr>
<th>Simulation Flow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NativeLink flow</td>
<td>The NativeLink automated flow supports a variety of design flows. Do not use NativeLink if you require direct control over every aspect of simulation.</td>
</tr>
<tr>
<td></td>
<td>• Use NativeLink to generate simulation scripts to compile your design and simulation libraries, and to automatically launch your simulator.</td>
</tr>
<tr>
<td></td>
<td>• Specify your own compilation, elaboration, and simulation scripts for testbench and simulation model files that have not been analyzed by the Quartus Prime software.</td>
</tr>
<tr>
<td></td>
<td>• Use NativeLink to supplement your scripts by automatically compiling design files, IP simulation model files, and Altera simulation library models.</td>
</tr>
<tr>
<td></td>
<td>• To use NativeLink for Arria 10 devices and later, you must add to your project the .qsys file generated for IP or Qsys system. To use NativeLink for all other device families, you must add to your project the .qip and .sip files generated for IP or Qsys systems.</td>
</tr>
<tr>
<td></td>
<td>• The Quartus Prime Pro Edition software does not support NativeLink RTL simulation</td>
</tr>
<tr>
<td>Custom flows</td>
<td>Custom flows support manual control of all aspects of simulation, including the following:</td>
</tr>
<tr>
<td></td>
<td>• Manually compile and simulate testbench, design, IP, and simulation model libraries, or write scripts to automate compilation and simulation in your simulator.</td>
</tr>
<tr>
<td></td>
<td>• Use the Simulation Library Compiler to compile simulation libraries for all Altera devices and supported third-party simulators and languages.</td>
</tr>
<tr>
<td></td>
<td>Use the custom flow if you require any of the following:</td>
</tr>
<tr>
<td></td>
<td>• Custom compilation commands for design, IP, or simulation library model files (for example, macros, debugging or optimization options, or other simulator-specific options).</td>
</tr>
<tr>
<td></td>
<td>• Multi-pass simulation flows.</td>
</tr>
<tr>
<td></td>
<td>• Flows that use dynamically generated simulation scripts.</td>
</tr>
<tr>
<td>Specialized flows</td>
<td>Altera supports specialized flows for various design variations, including the following:</td>
</tr>
<tr>
<td></td>
<td>• For simulation of Altera example designs, refer to the documentation for the example design or to the IP core user guide.</td>
</tr>
<tr>
<td></td>
<td>• For simulation of Qsys designs, refer to <em>Creating a System with Qsys</em>.</td>
</tr>
<tr>
<td></td>
<td>• For simulation of designs that include the Nios II embedded processor, refer to <em>Simulating a Nios II Embedded Processor</em>.</td>
</tr>
</tbody>
</table>
Preparing for Simulation

Preparing for RTL or gate-level simulation involves compiling the RTL or gate-level representation of your design and testbench. You must also compile IP simulation models, models from the Altera simulation libraries, and any other model libraries required for your design.

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 1-5: IP Simulation Script Utilities

<table>
<thead>
<tr>
<th>Utility</th>
<th>Description</th>
<th>Syntax</th>
<th>Generated Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip-setup-simulation</td>
<td>Finds all Altera IP cores in your project and automates regeneration of a combined simulation script after upgrading software or IP versions.</td>
<td>ip-setup-simulation --quartus-project=&lt;project&gt;.qpf --output-directory=&lt;directory&gt;</td>
<td>N/A</td>
</tr>
</tbody>
</table>
| 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 1-1: Incorporating IP Simulation Scripts into Top-Level Script
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
   ```
Incorporating Cadence IP Simulation Scripts

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

```bash
# # 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):

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

```sh
# 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:

```sh
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`.

```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:

```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

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 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.

## Compiling Simulation Models

The Quartus Prime software includes simulation models for all Altera IP cores. These models include IP functional simulation models, and device family-specific models in the Quartus Prime installation path/eda/sim_lib directory. These models include IEEE encrypted Verilog HDL models for both Verilog HDL and VHDL simulation.

Before running simulation, you must compile the appropriate simulation models from the Altera simulation libraries using any of the following methods:

- Use the NativeLink feature to automatically compile your design, Altera IP, simulation model libraries, and testbench.
- Run the Simulation Library Compiler to compile all RTL and gate-level simulation model libraries for your device, simulator, and design language.
- Compile Altera simulation models manually with your simulator.

After you compile the simulation model libraries, you can reuse these libraries in subsequent simulations.

**Note:** The specified timescale precision must be within 1ps when using Altera simulation models.

## Related Information

### Altera Simulation Models

### Generating IP Simulation Files for RTL Simulation

The Quartus Prime software supports both Verilog HDL and VHDL simulation of encrypted and unencrypted Altera IP cores. If your design includes Altera IP cores, you must compile any corresponding IP simulation models in your simulator with the rest of your design and testbench. The Quartus Prime software generates and copies the simulation models for IP cores to your project directory.
You can use the following files to simulate your Altera IP variation.

### Table 1-6: Altera IP Simulation Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator setup script</td>
<td>Simulator-specific script to compile, elaborate, and simulate Altera IP models and simulation model library files. Copy the commands into your simulation script, or edit these files to compile, elaborate, and simulate your design and testbench.</td>
<td>Cadence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>cds.lib</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>ncsim_setup.sh</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>hdl.var</code></td>
</tr>
<tr>
<td></td>
<td>Mentor Graphics</td>
<td>• <code>msim_setup.tcl</code></td>
</tr>
<tr>
<td></td>
<td>Synopsys</td>
<td>• <code>synopsys_sim.setup</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>vcs_setup.sh</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>vcsmx_setup.sh</code></td>
</tr>
<tr>
<td></td>
<td>Aldec</td>
<td>• <code>rivierapro_setup.tcl</code></td>
</tr>
<tr>
<td>Simulation IP File (.sip) or Qsys System File (.qsys)</td>
<td>The .sip and .qsys files contain IP core simulation library mapping information. To use NativeLink for Arria 10 devices and later, you must add the .qsys file generated for IP or Qsys system to your project. To use NativeLink for all other device families, you must add the .qip and .sip files generated for IP or Qsys systems to your project.</td>
<td>&lt;design name&gt;.sip</td>
</tr>
<tr>
<td>IP functional simulation models</td>
<td>IP functional simulation models are cycle-accurate VHDL or Verilog HDL models generated by the Quartus Prime software for some Altera IP cores. IP functional simulation models support fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.</td>
<td>&lt;my_ip&gt;.vho</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;my_ip&gt;.vo</td>
</tr>
<tr>
<td>IEEE encrypted models</td>
<td>Arria V, Cyclone V, Stratix V, and newer simulation model libraries and IP simulation models are provided in Verilog HDL and IEEE encrypted Verilog HDL. VHDL simulation of these models is supported using your simulator’s co-simulation capabilities. IEEE encrypted Verilog HDL models are significantly faster than IP functional simulation models.</td>
<td>&lt;my_ip&gt;.v</td>
</tr>
</tbody>
</table>

### Generating IP Functional Simulation Models for RTL Simulation

Altera provides IP functional simulation models for some Altera IP cores. To generate IP functional simulation models, follow these steps:

- Turn on the **Generate Simulation Model** option when parameterizing the IP core.
- When you simulate your design, compile only the .vo or .vho for these IP cores in your simulator. In this case you should not compile the corresponding HDL file. The encrypted HDL file supports synthesis by only the Quartus Prime software.
Running a Simulation (NativeLink Flow)

The NativeLink feature integrates your EDA simulator with the Quartus Prime software and automates the following simulation steps:

- Set and reuse simulation settings
- Generate simulator-specific files and simulation scripts
- Compile Altera simulation libraries
- Launch your simulator automatically following Quartus Prime Analysis & Elaboration, Analysis & Synthesis, or after a full compilation.

Note: To use NativeLink for Arria 10 devices and later, you must add to your project the .qsys file generated for IP or Qsys system. To use NativeLink for all other device families, you must add to your project the .qip and .sip files generated for IP or Qsys systems.

Setting Up Simulation (NativeLink Flow)

Before running simulation using the NativeLink flow, you must specify settings for your simulator in the Quartus Prime software. To specify simulation settings in the Quartus Prime software, follow these steps:

1. Open a Quartus Prime project.
2. Click Tools > Options and specify the location of your simulator executable file.

Table 1-7: Execution Paths for EDA Simulators

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mentor Graphics</td>
<td>&lt;drive letter&gt;:&lt;simulator install path&gt;\win32aloem (Windows)</td>
</tr>
<tr>
<td>ModelSim-Altera</td>
<td>/&lt;simulator install path&gt;/bin (Linux)</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td>&lt;drive letter&gt;:&lt;simulator install path&gt;\win32 (Windows)</td>
</tr>
<tr>
<td>ModelSim</td>
<td>&lt;simulator install path&gt;/bin (Linux)</td>
</tr>
<tr>
<td>Mentor Graphics</td>
<td></td>
</tr>
<tr>
<td>QuestaSim</td>
<td></td>
</tr>
<tr>
<td>Synopsys VCS/VCS MX</td>
<td>&lt;simulator install path&gt;/bin (Linux)</td>
</tr>
<tr>
<td>Cadence Incisive Enterprise</td>
<td>&lt;simulator install path&gt;/tools/bin (Linux)</td>
</tr>
</tbody>
</table>
### Simulator Path

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aldec Active-HDL</td>
<td><code>&lt;drive letter&gt;:\&lt;simulator install path&gt;\bin</code> (Windows)</td>
</tr>
<tr>
<td>Aldec Riviera-PRO</td>
<td><code>&lt;simulator install path&gt;/bin</code> (Linux)</td>
</tr>
</tbody>
</table>

3. Click Assignments > Settings and specify options on the Simulation page and More NativeLink Settings dialog box. Specify default options for simulation library compilation, netlist and tool command script generation, and for launching RTL or gate-level simulation automatically following Quartus Prime processing.

4. If your design includes a testbench, turn on Compile test bench and then click Test Benches to specify options for each testbench. Alternatively, turn on Use script to compile testbench and specify the script file.

5. If you want to use a script to setup simulation, turn on Use script to setup simulation.

### Running RTL Simulation (NativeLink Flow)

To run RTL simulation using the NativeLink flow, follow these steps:

1. Set up the simulation environment.
2. Click Processing > Start > Analysis and Elaboration.
3. Click Tools > Run Simulation Tool > RTL Simulation.

   NativeLink compiles simulation libraries and launches and runs your RTL simulator automatically according to the NativeLink settings.

4. Review and analyze the simulation results in your simulator. Correct any functional errors in your design. If necessary, re-simulate the design to verify correct behavior.

### Running Gate-Level Simulation (NativeLink Flow)

To run gate-level simulation with the NativeLink flow, follow these steps:

1. Prepare for simulation.
2. Set up the simulation environment. To generate only a functional (rather than timing) gate-level netlist, click More EDA Netlist Writer Settings, and turn on Generate netlist for functional simulation only.
3. To synthesize the design, follow one of these steps:
   - To generate a post-fit functional or post-fit timing netlist and then automatically simulate your design according to your NativeLink settings, Click Processing > Start Compilation. Skip to step 6.
   - To synthesize the design for post-synthesis functional simulation only, click Processing > Start > Start Analysis and Synthesis.
4. To generate the simulation netlist, click Start EDA Netlist Writer.
5. Click Tools > Run Simulation Tool > Gate Level Simulation.
6. Review and analyze the simulation results in your simulator. Correct any unexpected or incorrect conditions found in your design. Simulate the design again until you verify correct behavior.
Running a Simulation (Custom Flow)

Use a custom simulation flow to support any of the following more complex simulation scenarios:

- Custom compilation, elaboration, or run commands for your design, IP, or simulation library model files (for example, macros, debugging/optimization options, simulator-specific elaboration or run-time options)
- Multi-pass simulation flows
- Flows that use dynamically generated simulation scripts

Use these to compile libraries and generate simulation scripts for custom simulation flows:

- NativeLink-generated scripts—use NativeLink only to generate simulation script templates to develop your own custom scripts.
- Simulation Library Compiler—compile Altera simulation libraries for your device, HDL, and simulator. Generate scripts to compile simulation libraries as part of your custom simulation flow. This tool does not compile your design, IP, or testbench files.
- IP and Qsys simulation scripts—use the scripts generated for Altera IP cores and Qsys systems as templates to create simulation scripts. If your design includes multiple IP cores or Qsys systems, you can combine the simulation scripts into a single script, manually or by using the `ip-make-simscript` utility.

Use the following steps in a custom simulation flow:

1. Compile the design and testbench files in your simulator.
2. Run the simulation in your simulator.

Post-synthesis and post-fit gate-level simulations run significantly slower than RTL simulation. Altera recommends that you verify your design using RTL simulation for functionality and use the TimeQuest timing analyzer for timing. Timing simulation is not supported for Arria V, Cyclone V, Stratix V, and newer families.

Related Information

Running EDA Simulators

Document Revision History

This document has the following revision history.
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2015.11.02   | 15.1.0  | • Added new Generating Version-Independent IP Simulation Scripts topic.  
• Added example IP simulation script templates for all supported simulators.  
• Added new Incorporating IP Simulation Scripts in Top-Level Scripts topic.  
• Updated simulator support table with latest version information.  
• Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04   | 15.0.0  | • Updated simulator support table with latest.  
• Gate-level timing simulation limited to Stratix IV and Cyclone IV devices.  
• Added mixed language simulation support in the ModelSim-Altera software. |
| 2014.06.30   | 14.0.0  | • Replaced MegaWizard Plug-In Manager information with IP Catalog. |
| May 2013     | 13.0.0  | • Updated introductory section and system and IP file locations. |
| November 2012| 12.1.0  | • Revised chapter to reflect latest changes to other simulation documentation. |
| June 2012    | 12.0.0  | • Reorganization of chapter to reflect various simulation flows.  
• Added NativeLink support for newer IP cores. |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| November 2011| 11.1.0  | • Added information about encrypted Altera simulation model files.  
|              |         | • Added information about IP simulation and NativeLink. |

**Related Information**

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

Comprehensive static timing analysis involves analysis of register-to-register, I/O, and asynchronous reset paths. Timing analysis with the TimeQuest Timing Analyzer uses data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations.

The TimeQuest analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing. This chapter is an overview of the concepts you need to know to analyze your designs with the TimeQuest analyzer.

Related Information
- The Quartus Prime TimeQuest Timing Analyzer on page 3-1
  For more information about the TimeQuest analyzer flow and TimeQuest examples.

TimeQuest Terminology and Concepts

Table 2-1: TimeQuest Analyzer Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>Most basic timing netlist unit. Used to represent ports, pins, and registers.</td>
</tr>
<tr>
<td>cells</td>
<td>Look-up tables (LUT), registers, digital signal processing (DSP) blocks, memory blocks, input/output elements, and so on. (1)</td>
</tr>
<tr>
<td>pins</td>
<td>Inputs or outputs of cells.</td>
</tr>
<tr>
<td>nets</td>
<td>Connections between pins.</td>
</tr>
<tr>
<td>ports</td>
<td>Top-level module inputs or outputs; for example, device pins.</td>
</tr>
<tr>
<td>clocks</td>
<td>Abstract objects representing clock domains inside or outside of your design.</td>
</tr>
</tbody>
</table>

(1) Some terms may be ambiguous without context.

ISO 9001:2008 Registered

© 2015 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRON, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
Notes:

1. For Stratix® devices, the LUTs and registers are contained in logic elements (LE) and modeled as cells.

---

### Timing Netlists and Timing Paths

The TimeQuest analyzer requires a timing netlist to perform timing analysis on any design. After you generate a timing netlist, the TimeQuest analyzer uses the data to help determine the different design elements in your design and how to analyze timing.

### The Timing Netlist

A sample design for which the TimeQuest analyzer generates a timing netlist equivalent.

**Figure 2-1: Sample Design**

The timing netlist for the sample design shows how different design elements are divided into cells, pins, nets, and ports.
Timing Paths

Timing paths connect two design nodes, such as the output of a register to the input of another register. Understanding the types of timing paths is important to timing closure and optimization. The TimeQuest analyzer uses the following commonly analyzed paths:

- **Edge paths**—connections from ports-to-pins, from pins-to-pins, and from pins-to-ports.
- **Clock paths**—connections from device ports or internally generated clock pins to the clock pin of a register.
- **Data paths**—connections from a port or the data output pin of a sequential element to a port or the data input pin of another sequential element.
- **Asynchronous paths**—connections from a port or asynchronous pins of another sequential element such as an asynchronous reset or asynchronous clear.

In addition to identifying various paths in a design, the TimeQuest analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You must constrain all clocks in your design before analyzing clock characteristics.
Data and Clock Arrival Times

After the TimeQuest analyzer identifies the path type, it can report data and clock arrival times at register pins.

The TimeQuest analyzer calculates data arrival time by adding the launch edge time to the delay from the clock source to the clock pin of the source register, the micro clock-to-output delay ($\mu t_{CO}$) of the source register, and the delay from the source register’s data output (Q) to the destination register’s data input (D).

The TimeQuest analyzer calculates data required time by adding the latch edge time to the sum of all delays between the clock port and the clock pin of the destination register, including any clock port buffer delays, and subtracts the micro setup time ($\mu t_{SU}$) of the destination register, where the $\mu t_{SU}$ is the intrinsic setup time of an internal register in the FPGA.

Figure 2-4: Data Arrival and Data Required Times

The basic calculations for data arrival and data required times including the launch and latch edges.

Figure 2-5: Data Arrival and Data Required Time Equations

\[
\text{Data Arrival Time} = \text{Launch Edge} + \text{Source Clock Delay} + \mu t_{CO} + \text{Register-to-Register Delay}
\]

\[
\text{Data Required Time} = \text{Latch Edge} + \text{Destination Clock Delay} - \mu t_{SU}
\]

Launch and Latch Edges

All timing relies on one or more clocks. In addition to analyzing paths, the TimeQuest analyzer determines clock relationships for all register-to-register transfers in your design.

The following figure shows the launch edge, which is the clock edge that sends data out of a register or other sequential element, and acts as a source for the data transfer. A latch edge is the active clock edge that captures data at the data port of a register or other sequential element, acting as a destination for the data transfer. In this example, the launch edge sends the data from register $\text{reg1}$ at 0 ns, and the register $\text{reg2}$ captures the data when triggered by the latch edge at 10 ns. The data arrives at the destination register before the next latch edge.
In timing analysis, and with the TimeQuest analyzer specifically, you create clock constraints and assign those constraints to nodes in your design. These clock constraints provide the structure required for repeatable data relationships. The primary relationships between clocks, in the same or different domains, are the setup relationship and the hold relationship.

**Note:** If you do not constrain the clocks in your design, the Quartus Prime software analyzes in terms of a 1 GHz clock to maximize timing based Fitter effort. To ensure realistic slack values, you must constrain all clocks in your design with real values.

**Clock Setup Check**

To perform a clock setup check, the TimeQuest analyzer determines a setup relationship by analyzing each launch and latch edge for each register-to-register path.

For each latch edge at the destination register, the TimeQuest analyzer uses the closest previous clock edge at the source register as the launch edge. The following figure shows two setup relationships, setup A and setup B. For the latch edge at 10 ns, the closest clock that acts as a launch edge is at 3 ns and is labeled setup A. For the latch edge at 20 ns, the closest clock that acts as a launch edge is 19 ns and is labeled setup B. TimQuest analyzes the most restrictive setup relationship, in this case setup B; if that relationship meets the design requirement, then setup A meets it by default.

**Figure 2-7: Setup Check**

The TimeQuest analyzer reports the result of clock setup checks as slack values. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met; negative slack indicates the margin by which a requirement is not met.

**Figure 2-8: Clock Setup Slack for Internal Register-to-Register Paths**

\[
\begin{align*}
\text{Clock Setup Slack} & = \text{Data Required Time} - \text{Data Arrival Time} \\
\text{Data Arrival Time} & = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu_{\text{COP}} + \text{Register-to-Register Delay} \\
\text{Data Required Time} & = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu_{\text{SOP}} - \text{Setup Uncertainty}
\end{align*}
\]
The TimeQuest analyzer performs setup checks using the maximum delay when calculating data arrival time, and minimum delay when calculating data required time.

**Figure 2-9: Clock Setup Slack from Input Port to Internal Register**

<table>
<thead>
<tr>
<th>Clock Setup Slack</th>
<th>= Data Required Time – Data Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Arrival Time</td>
<td>= Launch Edge + Clock Network Delay + Input Maximum Delay + Port-to-Register Delay</td>
</tr>
<tr>
<td>Data Required Time</td>
<td>= Latch Edge + Clock Network Delay to Destination Register – $\mu_{\text{ts}}$ – Setup Uncertainty</td>
</tr>
</tbody>
</table>

**Figure 2-10: Clock Setup Slack from Internal Register to Output Port**

<table>
<thead>
<tr>
<th>Clock Setup Slack</th>
<th>= Data Required Time – Data Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Required Time</td>
<td>= Latch Edge + Clock Network Delay to Output Port – Output Maximum Delay</td>
</tr>
<tr>
<td>Data Arrival Time</td>
<td>= Launch Edge + Clock Network Delay to Source Register + $\mu_{\text{to}}$ + Register-to-Port Delay</td>
</tr>
</tbody>
</table>

**Clock Hold Check**

To perform a clock hold check, the TimeQuest analyzer determines a hold relationship for each possible setup relationship that exists for all source and destination register pairs. The TimeQuest analyzer checks all adjacent clock edges from all setup relationships to determine the hold relationships.

The TimeQuest analyzer performs two hold checks for each setup relationship. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. From the possible hold relationships, the TimeQuest analyzer selects the hold relationship that is the most restrictive. The most restrictive hold relationship is the hold relationship with the smallest difference between the latch and launch edges and determines the minimum allowable delay for the register-to-register path. In the following example, the TimeQuest analyzer selects hold check A2 as the most restrictive hold relationship of two setup relationships, setup A and setup B, and their respective hold checks.

**Figure 2-11: Setup and Hold Check Relationships**

**Figure 2-12: Clock Hold Slack for Internal Register-to-Register Paths**

<table>
<thead>
<tr>
<th>Clock Hold Slack</th>
<th>= Data Arrival Time – Data Required Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Arrival Time</td>
<td>= Launch Edge + Clock Network Delay to Source Register + $\mu_{\text{to}}$ + Register-to-Register Delay</td>
</tr>
<tr>
<td>Data Required Time</td>
<td>= Latch Edge + Clock Network Delay to Destination Register + $\mu_{\text{ts}}$ + Hold Uncertainty</td>
</tr>
</tbody>
</table>
The TimeQuest analyzer performs hold checks using the minimum delay when calculating data arrival time, and maximum delay when calculating data required time.

**Figure 2-13: Clock Hold Slack Calculation from Input Port to Internal Register**

\[
\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}
\]

- **Data Arrival Time** = Launch Edge + Clock Network Delay + Input Minimum Delay + Pin-to-Register Delay
- **Data Required Time** = Latch Edge + Clock Network Delay to Destination Register + \( \mu \)

**Figure 2-14: Clock Hold Slack Calculation from Internal Register to Output Port**

\[
\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}
\]

- **Data Arrival Time** = Latch Edge + Clock Network Delay to Source Register + \( \mu \) + Register-to-Pin Delay
- **Data Required Time** = Latch Edge + Clock Network Delay – Output Minimum Delay

**Recovery and Removal Time**

Recovery time is the minimum length of time for the deassertion of an asynchronous control signal relative to the next clock edge.

For example, signals such as clear and preset must be stable before the next active clock edge. The recovery slack calculation is similar to the clock setup slack calculation, but it applies to asynchronous control signals.

**Figure 2-15: Recovery Slack Calculation if the Asynchronous Control Signal is Registered**

\[
\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}
\]

- **Data Required Time** = Latch Edge + Clock Network Delay to Destination Register – \( \mu \)
- **Data Arrival Time** = Launch Edge + Clock Network Delay to Source Register + \( \mu \) + Register-to-Register Delay

**Figure 2-16: Recovery Slack Calculation if the Asynchronous Control Signal is not Registered**

\[
\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}
\]

- **Data Required Time** = Latch Edge + Clock Network Delay to Destination Register – \( \mu \)
- **Data Arrival Time** = Launch Edge + Clock Network Delay + Input Maximum Delay + Port-to-Register Delay

**Note:** If the asynchronous reset signal is from a device I/O port, you must create an input delay constraint for the asynchronous reset port for the TimeQuest analyzer to perform recovery analysis on the path.

Removal time is the minimum length of time the deassertion of an asynchronous control signal must be stable after the active clock edge. The TimeQuest analyzer removal slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals.
Figure 2-17: Removal Slack Calculation if the Asynchronous Control Signal is Registered

Removal Slack Time = Data Arrival Time – Data Required Time  
Data Arrival Time = Launch Edge + Clock Network Delay to Source Register + μts of Source Register + Register-to-Register Delay  
Data Required Time = Latch Edge + Clock Network Delay to Destination Register + μth

Figure 2-18: Removal Slack Calculation if the Asynchronous Control Signal is not Registered

Removal Slack Time = Data Arrival Time – Data Required Time  
Data Arrival Time = Launch Edge + Clock Network Delay + Input Minimum Delay of Pin + Minimum Pin-to-Register Delay  
Data Required Time = Latch Edge + Clock Network Delay to Destination Register + μth

If the asynchronous reset signal is from a device pin, you must assign the **Input Minimum Delay** timing assignment to the asynchronous reset pin for the TimeQuest analyzer to perform removal analysis on the path.

**Multicycle Paths**

Multicycle paths are data paths that require a non-default setup and/or hold relationship for proper analysis.

For example, a register may be required to capture data on every second or third rising clock edge. An example of a multicycle path between the input registers of a multiplier and an output register where the destination latches data on every other clock edge.

Figure 2-19: Multicycle Path

A register-to-register path used for the default setup and hold relationship, the respective timing diagrams for the source and destination clocks, and the default setup and hold relationships, when the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns. The default setup relationship is 5 ns; the default hold relationship is 0 ns.
To accommodate the system requirements you can modify the default setup and hold relationships with a multicycle timing exception.

The actual setup relationship after you apply a multicycle timing exception. The exception has a multicycle setup assignment of two to use the second occurring latch edge; in this example, to 10 ns from the default value of 5 ns.

Related Information
- The Quartus Prime TimeQuest Timing Analyzer on page 3-1
  For more information about creating exceptions with multicycle paths.

Metastability

Metastability problems can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains because the designer cannot guarantee that the signal will meet setup and hold time requirements.

To minimize the failures due to metastability, circuit designers typically use a sequence of registers, also known as a synchronization register chain, or synchronizer, in the destination clock domain to resynchronize the data signals to the new clock domain.

The mean time between failures (MTBF) is an estimate of the average time between instances of failure due to metastability.
The TimeQuest analyzer analyzes the potential for metastability in your design and can calculate the MTBF for synchronization register chains. The MTBF of the entire design is then estimated based on the synchronization chains it contains.

In addition to reporting synchronization register chains found in the design, the Quartus Prime software also protects these registers from optimizations that might negatively impact MTBF, such as register duplication and logic retiming. The Quartus Prime software can also optimize the MTBF of your design if the MTBF is too low.

Related Information
- **Understanding Metastability in FPGAs**
  For more information about metastability, its effects in FPGAs, and how MTBF is calculated.
- **Managing Metastability with the Quartus Prime Software**
  For more information about metastability analysis, reporting, and optimization features in the Quartus Prime software.

### Common Clock Path Pessimism Removal

Common clock path pessimism removal accounts for the minimum and maximum delay variation associated with common clock paths during static timing analysis by adding the difference between the maximum and minimum delay value of the common clock path to the appropriate slack equation.

Minimum and maximum delay variation can occur when two different delay values are used for the same clock path. For example, in a simple setup analysis, the maximum clock path delay to the source register is used to determine the data arrival time. The minimum clock path delay to the destination register is used to determine the data required time. However, if the clock path to the source register and to the destination register share a common clock path, both the maximum delay and the minimum delay are used to model the common clock path during timing analysis. The use of both the minimum delay and maximum delay results in an overly pessimistic analysis since two different delay values, the maximum and minimum delays, cannot be used to model the same clock path.

**Figure 2-22: Typical Register to Register Path**

Segment A is the common clock path between \( \text{reg1} \) and \( \text{reg2} \). The minimum delay is 5.0 ns; the maximum delay is 5.5 ns. The difference between the maximum and minimum delay value equals the common clock path pessimism removal value; in this case, the common clock path pessimism is 0.5 ns. The TimeQuest analyzer adds the common clock path pessimism removal value to the appropriate slack equation to determine overall slack. Therefore, if the setup slack for the register-to-register path in the example equals 0.7 ns without common clock path pessimism removal, the slack would be 1.2 ns with common clock path pessimism removal.

You can also use common clock path pessimism removal to determine the minimum pulse width of a register. A clock signal must meet a register’s minimum pulse width requirement to be recognized by the register. A minimum high time defines the minimum pulse width for a positive-edge triggered register. A minimum low time defines the minimum pulse width for a negative-edge triggered register.
Clock pulses that violate the minimum pulse width of a register prevent data from being latched at the data pin of the register. To calculate the slack of the minimum pulse width, the TimeQuest analyzer subtracts the required minimum pulse width time from the actual minimum pulse width time. The TimeQuest analyzer determines the actual minimum pulse width time by the clock requirement you specified for the clock that feeds the clock port of the register. The TimeQuest analyzer determines the required minimum pulse width time by the maximum rise, minimum rise, maximum fall, and minimum fall times.

**Figure 2-23: Required Minimum Pulse Width time for the High and Low Pulse**

With common clock path pessimism, the minimum pulse width slack can be increased by the smallest value of either the maximum rise time minus the minimum rise time, or the maximum fall time minus the minimum fall time. In the example, the slack value can be increased by 0.2 ns, which is the smallest value between 0.3 ns (0.8 ns – 0.5 ns) and 0.2 ns (0.9 ns – 0.7 ns).

**Related Information**

-TimeQuest Timing Analyzer Page (Settings Dialog Box)
For more information, refer to the Quartus Prime Help.

**Clock-As-Data Analysis**

The majority of FPGA designs contain simple connections between any two nodes known as either a data path or a clock path.

A data path is a connection between the output of a synchronous element to the input of another synchronous element.

A clock is a connection to the clock pin of a synchronous element. However, for more complex FPGA designs, such as designs that use source-synchronous interfaces, this simplified view is no longer sufficient. Clock-as-data analysis is performed in circuits with elements such as clock dividers and DDR source-synchronous outputs.

The connection between the input clock port and output clock port can be treated either as a clock path or a data path. A design where the path from port `clk_in` to port `clk_out` is both a clock and a data path. The clock path is from the port `clk_in` to the register `reg_data` clock pin. The data path is from port `clk_in` to the port `clk_out`. 
With clock-as-data analysis, the TimeQuest analyzer provides a more accurate analysis of the path based on user constraints. For the clock path analysis, any phase shift associated with the phase-locked loop (PLL) is taken into consideration. For the data path analysis, any phase shift associated with the PLL is taken into consideration rather than ignored.

The clock-as-data analysis also applies to internally generated clock dividers. An internally generated clock divider. In this figure, waveforms are for the inverter feedback path, analyzed during timing analysis. The output of the divider register is used to determine the launch time and the clock port of the register is used to determine the latch time.

**Figure 2-25: Clock Divider**

Multicycle Clock Setup Check and Hold Check Analysis

You can modify the setup and hold relationship when you apply a multicycle exception to a register-to-register path.
Multicycle Clock Setup

The setup relationship is defined as the number of clock periods between the latch edge and the launch edge. By default, the TimeQuest analyzer performs a single-cycle path analysis, which results in the setup relationship being equal to one clock period (latch edge – launch edge). Applying a multicycle setup assignment, adjusts the setup relationship by the multicycle setup value. The adjustment value may be negative.

An end multicycle setup assignment modifies the latch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default latch edge. The following figure shows various values of the end multicycle setup (EMS) assignment and the resulting latch edge.

Figure 2-27: End Multicycle Setup Values

A start multicycle setup assignment modifies the launch edge of the source clock by moving the launch edge the specified number of clock periods to the left of the determined default launch edge. A start multicycle setup (SMS) assignment with various values can result in a specific launch edge.
The setup relationship reported by the TimeQuest analyzer for the negative setup relationship.

**Figure 2-28: Start Multicycle Setup Values**

The setup relationship is defined as the number of clock periods between the launch edge and the latch edge.

By default, the TimeQuest analyzer performs a single-cycle path analysis, which results in the hold relationship being equal to one clock period (launch edge – latch edge). When analyzing a path, the TimeQuest analyzer performs two hold checks. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. The TimeQuest analyzer reports only the most restrictive hold check. The TimeQuest analyzer calculates the hold check by comparing launch and latch edges.

The calculation the TimeQuest analyzer performs to determine the hold check.

**Figure 2-30: Hold Check**

\[
\text{hold check 1} = \text{current launch edge} - \text{previous latch edge} \\
\text{hold check 2} = \text{next launch edge} - \text{current latch edge}
\]

**Tip:** If a hold check overlaps a setup check, the hold check is ignored.
A start multicycle hold assignment modifies the launch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default launch edge. The following figure shows various values of the start multicycle hold (SMH) assignment and the resulting launch edge.

**Figure 2-31: Start Multicycle Hold Values**

An end multicycle hold assignment modifies the latch edge of the destination clock by moving the latch edge the specified number of clock periods to the left of the determined default latch edge. The following figure shows various values of the end multicycle hold (EMH) assignment and the resulting latch edge.

**Figure 2-32: End Multicycle Hold Values**
The hold relationship reported by the TimeQuest analyzer for the negative hold relationship shown in the figure above would look like this:

**Figure 2-33: End Multicycle Hold Values Reported by the TimeQuest Analyzer**

![Figure 2-33: End Multicycle Hold Values Reported by the TimeQuest Analyzer](image)

## Multicorner Analysis

The TimeQuest analyzer performs multicorner timing analysis to verify your design under a variety of operating conditions—such as voltage, process, and temperature—while performing static timing analysis.

To change the operating conditions or speed grade of the device used for timing analysis, use the `set_operating_conditions` command.

If you specify an operating condition Tcl object, the `-model`, `-speed`, `-temperature`, and `-voltage` options are optional. If you do not specify an operating condition Tcl object, the `-model` option is required; the `-speed`, `-temperature`, and `-voltage` options are optional.

**Tip:** To obtain a list of available operating conditions for the target device, use the `get_available_operating_conditions -all` command.

To ensure that no violations occur under various conditions during the device operation, perform static timing analysis under all available operating conditions.

### Table 2-2: Operating Conditions for Slow and Fast Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Speed Grade</th>
<th>Voltage</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td>Slowest speed grade in device density</td>
<td>$V_{cc}$ minimum supply ((1))</td>
<td>Maximum $T_J ((1))$</td>
</tr>
<tr>
<td>Fast</td>
<td>Fastest speed grade in device density</td>
<td>$V_{cc}$ maximum supply ((1))</td>
<td>Minimum $T_J ((1))$</td>
</tr>
</tbody>
</table>

**Note:**

1. Refer to the DC & Switching Characteristics chapter of the applicable device Handbook for $V_{cc}$ and $T_J$ values

In your design, you can set the operating conditions for to the slow timing model, with a voltage of 1100 mV, and temperature of 85°C with the following code:

```
set_operating_conditions -model slow -temperature 85 -voltage 1100
```
You can set the same operating conditions with a Tcl object:

```
set_operating_conditions 3_slow_1100mv_85c
```

The following block of code shows how to use the `set_operating_conditions` command to generate different reports for various operating conditions.

### Example 2-1: Script Excerpt for Analysis of Various Operating Conditions

```bash
# Specify initial operating conditions
set_operating_conditions -model slow -speed 3 -grade c -temperature 85 - voltage 1100
# Update the timing netlist with the initial conditions
update_timing_netlist
# Perform reporting
# Change initial operating conditions. Use a temperature of 0 C
set_operating_conditions -model slow -speed 3 -grade c -temperature 0 - voltage 1100
# Update the timing netlist with the new operating condition
update_timing_netlist
# Perform reporting
# Change initial operating conditions. Use a temperature of 0 C and a model of fast
set_operating_conditions -model fast -speed 3 -grade c -temperature 0 - voltage 1100
# Update the timing netlist with the new operating condition
update_timing_netlist
# Perform reporting
```

### Related Information

- `set_operating_conditions`

- `get_available_operating_conditions`

For more information about the `get_available_operating_conditions` command.

---

### Document Revision History

**Table 2-3: Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2014.12.15</td>
<td>14.1.0</td>
<td>Moved Multicycle Clock Setup Check and Hold Check Analysis section from the TimeQuest Timing Analyzer chapter.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated format</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Added social networking icons, minor text updates</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
The Quartus Prime TimeQuest Timing Analyzer

The Quartus Prime TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology. Use the TimeQuest analyzer GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

This document is organized to allow you to refer to specific subjects relating to the TimeQuest analyzer and timing analysis. The sections cover the following topics:

Enhanced Timing Analysis for Arria 10 on page 3-2
The TimeQuest Timing Analyzer supports new timing algorithms for the Arria® 10 device family which significantly improve the speed of the analysis.

Recommended Flow on page 3-2
The Quartus Prime TimeQuest analyzer performs constraint validation to timing verification as part of the compilation flow.

Timing Constraints on page 3-6
Timing analysis in the Quartus Prime software with the TimeQuest Timing Analyzer relies on constraining your design to make it meet your timing requirements.

Running the TimeQuest Analyzer on page 3-54
When you compile a design, the TimeQuest timing analyzer automatically performs multi-corner signoff timing analysis after the Fitter has finished.

Understanding Results on page 3-57
Knowing how your constraints are displayed when analyzing a path is one of the most important skills of timing analysis.

Constraining and Analyzing with Tcl Commands on page 3-65
You can use Tcl commands from the Quartus Prime software Tcl Application Programming Interface (API) to constrain, analyze, and collect information for your design.

Generating Timing Reports on page 3-70
The TimeQuest analyzer provides real-time static timing analysis result reports.

Document Revision History on page 3-71
Related Information

- **Timing Analysis Overview** on page 2-1
  For more information about basic timing analysis concepts and how they pertain to the TimeQuest analyzer.
- **TimeQuest Timing Analyzer Resource Center**
  For more information about Altera resources available for the TimeQuest analyzer.
- **Altera Training**
  For more information about the TimeQuest analyzer.

Enhanced Timing Analysis for Arria 10

The TimeQuest Timing Analyzer supports new timing algorithms for the Arria® 10 device family which significantly improve the speed of the analysis.

These algorithms are enabled by default for Arria 10 devices, and can be enabled for earlier families with an assignment. The new analysis engine analyzes the timing graph a fixed number of times. Previous TimeQuest analysis analyzed the timing graph as many times as there were constraints in your Synopsys Design Constraint (SDC) file.

Recommended Flow for First Time Users

The Quartus Prime TimeQuest analyzer performs constraint validation to timing verification as part of the compilation flow. Both the TimeQuest analyzer and the Fitter use of constraints contained in a Synopsys Design Constraints (.sdc) file. The following flow is recommended if you have not created a project and do not have a SDC file with timing constraints for your design.

**Figure 3-1: Design Flow with the TimeQuest Timing Analyzer**

Creating and Setting Up your Design

You must first create your project in the Quartus Prime software. Include all the necessary design files, including any existing Synopsys Design Constraints (.sdc) files, also referred to as SDC files, that contain
timing constraints for your design. Some reference designs, or Altera or partner IP cores may already include one or more SDC files.

All SDC files must be added to your project so that your constraints are processed when the Quartus Prime software performs Fitting and Timing Analysis. Typically you must create an SDC file to constrain your design.

Related Information
SDC File Precedence on page 3-56

Specifying Timing Requirements

Before running timing analysis with the TimeQuest analyzer, you must specify timing constraints, describe the clock frequency requirements and other characteristics, timing exceptions, and I/O timing requirements of your design. You can use the TimeQuest Timing Analyzer Wizard to enter initial constraints for your design, and then refine timing constraints with the TimeQuest analyzer GUI.

Both the TimeQuest analyzer and the Fitter use of constraints contained in a Synopsis Design Constraints (.sdc) file.

The constraints in the SDC file are read in sequence. You must first make a constraint before making any references to that constraint. For example, if a generated clock references a base clock, the base clock constraint must be made before the generated clock constraint.

If you are new to timing analysis with the TimeQuest analyzer, you can use template files included with the Quartus Prime software and the interactive dialog boxes to create your initial SDC file. To use this method, refer to Performing an Initial Analysis and Synthesis.

If you are familiar with timing analysis, you can also create an SDC file in your preferred text editor. Don’t forget to include the SDC file in the project when you are finished.

Related Information

- Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor on page 3-4
  For more information on using the Text Editor templates for SDC constraints.
- Identifying the Quartus Prime Software Executable from the SDC File on page 3-69

Performing an Initial Analysis and Synthesis

Perform Analysis and Synthesis on your design so that you can find design entry names in the Node Finder to simplify creating constraints.

The Quartus Prime software populates an internal database with design element names. You must synthesize your design in order for the Quartus Prime software to assign names to your design elements, for example, pins, nodes, hierarchies, and timing paths.

If you have already compiled your design, you do not need to perform the synthesis step again, because compiling the design automatically performs synthesis. You can either perform Analysis and Synthesis to create a post-map database, or perform a full compilation to create a post-fit database. Creating a post-map database is faster than a post-fit database, and is sufficient for creating initial timing constraints.
Note: When compiling for the Arria 10 device family, the following commands are required to perform initial synthesis and enable you to use the Node Finder to find names in your design:

```plaintext
quartus_map <design>
quartus_fit <design> --floorplan
quartus_sta <design> --post_map
```

When compiling for other devices, you can exclude the `quartus_fit <design> --floorplan` step:

```plaintext
quartus_map <design>
quartus_sta <design> --post_map
```

Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor

You can create an SDC file from constraint templates in the Quartus Prime software with the Quartus Prime Text Editor, or with your preferred text editor.

1. On the File menu, click New.
2. In the New dialog box, select the Synopsys Design Constraints File type from the Other Files group. Click OK.
3. Right-click in the blank SDC file in the Quartus Prime Text Editor, then click Insert Constraint. Choose Clock Constraint followed by Set Clock Groups since they are the most widely used constraints.

The Quartus Prime Text Editor displays a dialog box with interactive fields for creating constraints. For example, the Create Clock dialog box shows you the waveform for your create_clock constraint while you adjust the Period and Rising and Falling waveform edge settings. The actual constraint is displayed in the SDC command field. Click Insert to use the constraint in your SDC.

or

4. Click the Insert Template button on the text editor menu, or, right-click in the blank SDC file in the Quartus Prime Text Editor, then click Insert Template TimeQuest.
   a. In the Insert Template dialog box, expand the TimeQuest section, then expand the SDC Commands section.
   b. Expand a command category, for example, Clocks.
   c. Select a command. The SDC constraint appears in the Preview pane.
   d. Click Insert to paste the SDC constraint into the blank SDC file you created in step 2. This creates a generic constraint for you to edit manually, replacing variables such as clock names, period, rising and falling edges, ports, etc.
5. Repeat as needed with other constraints, or click Close to close the Insert Template dialog box.

You can now use any of the standard features of the Quartus Prime Text Editor to modify the SDC file, or save the SDC file to edit in a text editor. Your SDC can be saved with the same name as the project, and generally should be stored in the project directory.

Related Information

- Create Clocks Dialog Box
- Set Clock Groups Dialog Box

For more information on Create Clocks and Set Clock Groups, refer to the Quartus Prime Help.
Performing a Full Compilation

After creating initial timing constraints, compile your design.

During a full compilation, the Fitter uses the TimeQuest analyzer repeatedly to perform timing analysis with your timing constraints. By default, the Fitter can stop early if it meets your timing requirements, instead of attempting to achieve the maximum performance. You can modify this by changing the Fitter effort settings in the Quartus Prime software.

If you are using the Quartus Prime Pro Edition software, you have some command line options available when performing iterative timing analysis on large designs. You can perform a less intensive analysis with `quartus_sta --mode=implement`. In this mode, the Quartus Prime software performs a reduced-corner timing analysis. When your modification iterations have achieved the desired result, you can use `quartus_sta --mode=finalize` to perform final Fitter optimizations and a full four-corner timing analysis.

Related Information

- **Analyzing Timing in Designs Compiled in Previous Versions** on page 3-6
  For more information about importing databases compiled in previous versions of the software.
- **Fitter Settings Page (Settings Dialog Box)**
  For more information about changing Fitter effort, refer to the Quartus Prime Help.

Verifying Timing

The TimeQuest analyzer examines the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as positive slack or negative slack. Negative slack indicates a timing violation. If you encounter violations along timing paths, use the timing reports to analyze your design and determine how best to optimize your design. If you modify, remove, or add constraints, you should perform a full compilation again. This iterative process helps resolve timing violations in your design.

There is a recommended flow for constraining and analyzing your design within the TimeQuest analyzer, and each part of the flow has a corresponding Tcl command.
Analyzing Timing in Designs Compiled in Previous Versions

Performing a full compilation can be a lengthy process, however, once your design meets timing you can export the design database for later use. This can include operations such as verification of subsequent timing models, or use in a later version of the Quartus Prime software.

When you re-open the project, the Quartus Prime software opens the exported database from the export directory. You can then run TimeQuest on the design without having to recompile the project.

To export the database in the previous version of the Quartus Prime software, click **Project > Export Database** and select the export directory to contain the exported database.

To import a database in a later version of the Quartus Prime software, click **File > Open** and select the Quartz Prime Project file (**.qpf**) for the project.

Once you import the database, you can perform any TimeQuest analyzer functions on the design without recompiling.

**Timing Constraints**

Timing analysis in the Quartus Prime software with the TimeQuest Timing Analyzer relies on constraining your design to make it meet your timing requirements. When discussing these constraints, they can be referred to as timing constraints, SDC constraints, or SDC commands interchangeably.

**Recommended Starting SDC Constraints**

Almost every beginning SDC file should contain the following four commands:
create_clock on page 3-7
derive_pll_clocks on page 3-8
derive_clock_uncertainty on page 3-8

SDC Constraint Creation Summary on page 3-8
Those are the first three steps, which can usually be done very quickly. For a sample design with two clocks coming into it, your SDC file might look like this example:

set_clock_groups on page 3-9

Related Information
Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor on page 3-4

create_clock
The first statements in a SDC file should be constraints for clocks, for example, constrain the external clocks coming into the FPGA with create_clock. An example of the basic syntax is:

create_clock -name sys_clk -period 8.0 [get_ports fpga_clk]

This command creates a clock called sys_clk with an 8ns period and applies it to the port called fpga_clk.

Note: Both Tcl files and SDC files are case-sensitive, so make sure references to pins, ports, or nodes, such as fpga_clk match the case used in your design.

By default, the clock has a rising edge at time 0ns, and a 50% duty cycle, hence a falling edge at time 4ns. If you require a different duty cycle or to represent an offset, use the -waveform option, however, this is seldom necessary.

It is common to create a clock with the same name as the port it is applied to. In the example above, this would be accomplished by:

create_clock -name fpga_clk -period 8.0 [get_ports fpga_clk]

There are now two unique objects called fpga_clk, a port in your design and a clock applied to that port.

Note: In Tcl syntax, square brackets execute the command inside them, so [get_ports fpga_clk] executes a command that finds all ports in the design that match fpga_clk and returns a collection of them. You can enter the command without using the get_ports collection command, as shown in the following example. There are benefits to using collection commands, which are described in "Collection Commands".

create_clock -name sys_clk -period 8.0 fpga_clk

Repeat this process, using one create_clock command for each known clock coming into your design. Later on you can use Report Unconstrained Paths to identify any unconstrained clocks.

Note: Rather than typing constraints, users can enter constraints through the GUI. After launching TimeQuest, open the SDC file from TimeQuest or Quartus Prime, place the cursor where the new constraint will go, and go to Edit > Insert Constraint, and choose the constraint.

Warning: Using the Constraints menu option in the TimeQuest GUI applies constraints directly to the timing database, but makes no entry in the SDC file. An advanced user may find reasons to do this, but if you are new to TimeQuest, Altera recommends entering your constraints directly into your SDC with the Edit > Insert Constraint command.
**Related Information**

Creating Base Clocks on page 3-11

dervie_pll_clocks

After the `create_clock` commands add the following command into your SDC file:

```
dervie_pll_clocks
```

This command automatically creates a generated clock constraint on each output of the PLLs in your design.

When PLLs are created, you define how each PLL output is configured. Because of this, the TimeQuest analyzer can automatically constrain them with the `derive_pll_clocks` command.

This command also creates other constraints. It constrains transceiver clocks. It adds multicycles between LVDS SERDES and user logic.

The `derive_pll_clocks` command prints an Info message to show each generated clock it creates.

If you are new to the TimeQuest analyzer, you may decide not to use `derive_pll_clocks`, and instead cut-and-paste each `create_generated_clock` assignment into the SDC file. There is nothing wrong with this, since the two are identical. The problem is that when you modify a PLL setting, you must remember to change its generated clock in the SDC file. Examples of this type of change include modifying an existing output clock, adding a new PLL output, or making a change to the PLL’s hierarchy. Too many designers forget to modify the SDC file and spend time debugging something that `derive_pll_clocks` would have updated automatically.

**Related Information**

- Creating Base Clocks on page 3-11
- Deriving PLL Clocks on page 3-18

dervie_clock_uncertainty

Add the following command to your SDC file:

```
dervie_clock_uncertainty
```

This command calculates clock-to-clock uncertainties within the FPGA, due to characteristics like PLL jitter, clock tree jitter, etc. This should be in all SDC files and the TimeQuest analyzer generates a warning if this command is not found in your SDC files.

**Related Information**

Accounting for Clock Effect Characteristics on page 3-21

**SDC Constraint Creation Summary**

Those are the first three steps, which can usually be done very quickly. For a sample design with two clocks coming into it, your SDC file might look like this example:

```
create_clock -period 20.00 -name adc_clk [get_ports adc_clk]
create_clock -period 8.00 -name sys_clk [get_ports sys_clk]
derive_pll_clocks
derive_clock_uncertainty
```
With the constraints discussed previously, most, if not all, of the clocks in the design are now constrained. In the TimeQuest analyzer, all clocks are related by default, and you must indicate which clocks are not related. For example, if there are paths between an 8ns clock and 10ns clock, even if the clocks are completely asynchronous, the TimeQuest analyzer attempts to meet a 2ns setup relationship between these clocks unless you indicate that they are not related. The TimeQuest analyzer analyzes everything known, rather than assuming that all clocks are unrelated and requiring that you relate them. The SDC language has a powerful constraint for setting unrelated clocks called `set_clock_groups`. A template for the typical use of the `set_clock_groups` command is:

```plaintext
set_clock_groups -asynchronous -group {<clock1>...<clockn>} ... 
   -group {<clocks>...<clockn>}
```

The `set_clock_groups` command does not actually group clocks. Since the TimeQuest analyzer assumes all clocks are related by default, all clocks are effectively in one big group. Instead, the `set_clock_groups` command cuts timing between clocks in different groups.

There is no limit to the number of times you can specify a group option with `-group {<group of clocks>}`. When entering constraints through the GUI with `Edit > Insert Constraint`, the **Set Clock Groups** dialog box only permits two clock groups, but this is only a limitation of that dialog box. You can always manually add more into the SDC file.

Any clock not listed in the assignment is related to all clocks. If you forget a clock, the TimeQuest analyzer acts conservatively and analyzes that clock in context with all other domains to which it connects.

The `set_clock_groups` command requires either the `-asynchronous` or `-exclusive` option. The `-asynchronous` flag means the clocks are both toggling, but not in a way that can synchronously pass data. The `-exclusive` flag means the clocks do not toggle at the same time, and hence are mutually exclusive. An example of this might be a clock multiplexor that has two generated clock assignments on its output. Since only one can toggle at a time, these clocks are `-exclusive`. TimeQuest does not currently analyze crosstalk explicitly. Instead, the timing models use extra guard bands to account for any potential crosstalk-induced delays. TimeQuest treats the `-asynchronous` and `-exclusive` options the same.

A clock cannot be within multiple `-group` groupings in a single assignment, however, you can have multiple `set_clock_groups` assignments.

Another way to cut timing between clocks is to use `set_false_path`. To cut timing between `sys_clk` and `dsp_clk`, a user might enter:

```plaintext
set_false_path -from [get_clocks sys_clk] -to [get_clocks dsp_clk]
set_false_path -from [get_clocks dsp_clk] -to [sys_clk]
```

This works fine when there are only a few clocks, but quickly grows to a huge number of assignments that are completely unreadable. In a simple design with three PLLs that have multiple outputs, the `set_clock_groups` command can clearly show which clocks are related in less than ten lines, while `set_false_path` may be over 50 lines and be very non-intuitive on what is being cut.

**Related Information**

- Creating Generated Clocks on page 3-15
- Relaxing Setup with `set_multicycle_path` on page 3-31
- Accounting for a Phase Shift on page 3-32
Tips for Writing a set_clock_groups Constraint

Since `derive_pll_clocks` creates many of the clock names, you may not know all of the clock names to use in the clock groups.

A quick way to make this constraint is to use the SDC file you have created so far, with the three basic constraints described in previous topics. Make sure you have added it to your project, then open the TimeQuest timing analyzer GUI.

In the Task panel of the TimeQuest analyzer, double-click on Report Clocks. This reads your existing SDC and applies it to your design, then reports all the clocks. From that report, highlight all of the clock names in the first column, and copy the names.

You have just copied all the clock names in your design in the exact format the TimeQuest analyzer recognizes. Paste them into your SDC file to make a list of all clock names, one per line.

Format that list into the `set_clock_groups` command by cutting and pasting clock names into appropriate groups. Then enter the following empty template in your SDC file:

```tcl
set_clock_groups -asynchronous -group { \\
 } \\
-group { \\
 } \\
-group { \\
 } \\
-group { \\
}
```

Cut and paste clocks into groups to define how they’re related, adding or removing groups as necessary. Format to make the code readable.

**Note:** This command can be difficult to read on a single line. Instead, you should make use of the Tcl line continuation character `\"`. By putting a space after your last character and then `\"`, the end-of-line character is escaped. (And be careful not to have any whitespace after the escape character, or else it will escape the whitespace, not the end-of-line character).

```tcl
set_clock_groups -asynchronous \\
-group {adc_clk \\
    the_adc_pll\altpll_component_autogenerated|pll|clk[0] \\
    the_adc_pll\altpll_component_autogenerated|pll|clk[1] \\
    the_adc_pll\altpll_component_autogenerated|pll|clk[2] \\
} \\
-group {sys_clk \\
    the_system_pll\altpll_component_autogenerated|pll|clk[0] \\
    the_system_pll\altpll_component_autogenerated|pll|clk[1] \\
} \\
-group {the_system_pll\altpll_component_autogenerated|pll|clk[2] \\
}
```

**Note:** The last group has a PLL output `system_pll..|clk[2]` while the input clock and other PLL outputs are in different groups. If PLLs are used, and the input clock frequency is not related to the frequency of the PLL’s outputs, they must be treated asynchronously. Usually most outputs of a PLL are related and hence in the same group, but this is not a requirement, and depends on the requirements of your design.

For designs with complex clocking, writing this constraint can be an iterative process. For example, a design with two DDR3 cores and high-speed transceivers could easily have thirty or more clocks. In those cases, you can just add the clocks you’ve created. Since clocks not in the command are still related to every clock, you are conservatively grouping what is known. If there are still failing paths in the design between unrelated clock domains, you can start adding in the new clock domains as necessary. In this case, a large number of the clocks won’t actually be in the `set_clock_groups` command, since they are either cut in
the SDC file for the IP core (such as the SDC files generated by the DDR3 cores), or they only connect to clock domains to which they are related.

For many designs, that is all that’s necessary to constrain the core. Some common core constraints that will not be covered in this quick start section that users do are:

- Add multicycles between registers which can be analyzed at a slower rate than the default analysis, in other words, increasing the time when data can be read, or ‘opening the window’. For example, a 10ns clock period will have a 10ns setup relationship. If the data changes at a slower rate, or perhaps the registers toggle at a slower rate due to a clock enable, then you should apply a multicycle that relaxes the setup relationship (opens the window so that valid data can pass). This is a multiple of the clock period, making the setup relationship 20ns, 40ns, etc., while keeping the hold relationship at 0ns. These types of multicycles are generally applied to paths.

- The second common form of multicycle is when the user wants to advance the cycle in which data is read, or ‘shift the window’. This generally occurs when your design performs a small phase-shift on a clock. For example, if your design has two 10ns clocks exiting a PLL, but the second clock has a 0.5ns phase-shift, the default setup relationship from the main clock to the phase-shifted clock is 0.5ns and the hold relationship is -9.5ns. It is almost impossible to meet a 0.5ns setup relationship, and most likely you intend the data to transfer in the next window. By adding a multicycle from the main clock to the phase-shifted clock, the setup relationship becomes 10.5ns and the hold relationship becomes 0.5ns. This multicycle is generally applied between clocks and is something the user should think about as soon as they do a small phase-shift on a clock. This type of multicycle is called shifting the window.

- Add a `create_generated_clock` to ripple clocks. When a register’s output drives the `clk` port of another register, that is a ripple clock. Clocks do not propagate through registers, so all ripple clocks must have a `create_generated_clock` constraint applied to them for correct analysis. Unconstrained ripple clocks appear in the `Report Unconstrained Paths` report, so they are easily recognized. In general, ripple clocks should be avoided for many reasons, and if possible, a clock enable should be used instead.

- Add a `create_generated_clock` to clock mux outputs. Without this, all clocks propagate through the mux and are related. TimeQuest analyze paths downstream from the mux where one clock input feeds the source register and the other clock input feeds the destination, and vice-versa. Although it could be valid, this is usually not preferred behavior. By putting `create_generated_clock` constraints on the mux output, which relates them to the clocks coming into the mux, you can correctly group these clocks with other clocks.

Creating Clocks and Clock Constraints

Clocks specify timing requirements for synchronous transfers and guide the Fitter optimization algorithms to achieve the best possible placement for your design. You must define all clocks and any associated clock characteristics, such as uncertainty or latency. The TimeQuest analyzer supports SDC commands that accommodate various clocking schemes such as:

- Base clocks
- Virtual clocks
- Multifrequency clocks
- Generated clocks

Creating Base Clocks

Base clocks are the primary input clocks to the device. Unlike clocks that are generated in the device (such as an on-chip PLL), base clocks are generated by off-chip oscillators or forwarded from an external device. Define base clocks at the top of your SDC file, because generated clocks and other constraints often reference base clocks. The TimeQuest timing analyzer ignores any constraints that reference a clock that has not been defined.
Use the `create_clock` command to create a base clock. Use other constraints, such as those described in *Accounting for Clock Effect Characteristics*, to specify clock characteristics such as uncertainty and latency.

The following examples show the most common uses of the `create_clock` constraint:

**create_clock Command**

To specify a 100 MHz requirement on a `clk_sys` input clock port you would enter the following in your SDC file:

```plaintext
create_clock -period 10 -name clk_sys [get_ports clk_sys]
```

**100 MHz Shifted by 90 Degrees Clock Creation**

This example creates a 10 ns clock with a 50% duty cycle that is phase shifted by 90 degrees applied to port `clk_sys`. This type of clock definition is most commonly used when the FPGA receives source synchronous, double-rate data that is center-aligned with respect to the clock.

```plaintext
create_clock -period 10 -waveform { 2.5 7.5 } [get_ports clk_sys]
```

**Two Oscillators Driving the Same Clock Port**

You can apply multiple clocks to the same target with the `-add` option. For example, to specify that the same clock input can be driven at two different frequencies, enter the following commands in your SDC file:

```plaintext
create_clock -period 10 -name clk_100 [get_ports clk_sys]
cREATE_CLOCK -PERIOD 5 -NAME CLK_200 [GET_PORTS CLK_SYS] -ADD
```

Although it is not common to have more than two base clocks defined on a port, you can define as many as are appropriate for your design, making sure you specify `-add` for all clocks after the first.

**Creating Multifrequency Clocks**

You must create a multifrequency clock if your design has more than one clock source feeding a single clock node in your design. The additional clock may act as a low-power clock, with a lower frequency than the primary clock. If your design uses multifrequency clocks, use the `set_clock_groups` command to define clocks that are exclusive.

To create multifrequency clocks, use the `create_clock` command with the `-add` option to create multiple clocks on a clock node. You can create a 10 ns clock applied to clock port `clk`, and then add an additional 15 ns clock to the same clock port. The TimeQuest analyzer uses both clocks when it performs timing analysis.

```plaintext
create_clock –period 10 –name clock_primary –waveform { 0 5 } \ [get_ports clk]
create_clock –period 15 –name clock_secondary –waveform { 0 7.5 } \ [get_ports clk] –add
```

**Related Information**

- *Accounting for Clock Effect Characteristics* on page 3-21
- `create_clock`
- `get_ports`
  
  For more information about these commands, refer to Quartus Prime Help.
Automatically Detecting Clocks and Creating Default Clock Constraints

To automatically create base clocks in your design, use the `derive_clocks` command. The `derive_clocks` command is equivalent to using the `create_clock` command for each register or port feeding the clock pin of a register. The `derive_clocks` command creates clock constraints on ports or registers to ensure every register in your design has a clock constraints, and it applies one period to all base clocks in your design.

You can have the TimeQuest analyzer create a base clock with a 100 Mhz requirement for unconstrained base clock nodes.

```
derive_clocks -period 10
```

**Warning:** Do not use the `derive_clocks` command for final timing sign-off; instead, you should create clocks for all clock sources with the `create_clock` and `create_generated_clock` commands. If your design has more than a single clock, the `derive_clocks` command constrains all the clocks to the same specified frequency. To achieve a thorough and realistic analysis of your design's timing requirements, you should make individual clock constraints for all clocks in your design.

If you want to have some base clocks created automatically, you can use the `-create_base_clocks` option to `derive_pll_clocks`. With this option, the `derive_pll_clocks` command automatically creates base clocks for each PLL, based on the input frequency information specified when the PLL was instantiated. The base clocks are named matching the port names. This feature works for simple port-to-PLL connections. Base clocks are not automatically generated for complex PLL connectivity, such as cascaded PLLs. You can also use the command `derive_pll_clocks -create_base_clocks` to create the input clocks for all PLL inputs automatically.

**Related Information**

`derive_clocks`
For more information about this command, refer to Quartus Prime Help.

Creating Virtual Clocks

A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design.

To create virtual clocks, use the `create_clock` command with no value specified for the `<targets>` option. This example defines a 100Mhz virtual clock because no target is specified.

```
create_clock -period 10 -name my_virt_clk
```

**I/O Constraints with Virtual Clocks**

Virtual clocks are most commonly used in I/O constraints; they represent the clock at the external device connected to the FPGA.

For the output circuit shown in the following figure, you should use a base clock to constrain the circuit in the FPGA, and a virtual clock to represent the clock driving the external device. Examples of the base clock, virtual clock, and output delay constraints for such a circuit are shown below.
You can create a 10 ns virtual clock named `virt_clk` with a 50% duty cycle where the first rising edge occurs at 0 ns by adding the following code to your SDC file. The virtual clock is then used as the clock source for an output delay constraint.

**Example 3-1: Virtual Clock**

```plaintext
# create base clock for the design
create_clock -period 5 [get_ports system_clk]
# create the virtual clock for the external register
create_clock -period 10 -name virt_clk
# set the output delay referencing the virtual clock
set_output_delay -clock virt_clk -max 1.5 [get_ports dataout]
set_output_delay -clock virt_clk -min 0.0 [get_ports dataout]
```

**Related Information**
- `set_input_delay`
- `set_output_delay`

For more information about these commands, refer to Quartus Prime Help.

**Example of Specifying an I/O Interface Clock**

To specify I/O interface uncertainty, you must create a virtual clock and constrain the input and output ports with the `set_input_delay` and `set_output_delay` commands that reference the virtual clock.

When the `set_input_delay` or `set_output_delay` commands reference a clock port or PLL output, the virtual clock allows the `derive_clock_uncertainty` command to apply separate clock uncertainties for internal clock transfers and I/O interface clock transfers.

Create the virtual clock with the same properties as the original clock that is driving the I/O port.
Example 3-2: SDC Commands to Constrain the I/O Interface

```
# Create the base clock for the clock port
create_clock -period 10 -name clk_in [get_ports clk_in]
# Create a virtual clock with the same properties of the base clock
# driving the source register
create_clock -period 10 -name virt_clki
# Create the input delay referencing the virtual clock and not the base
# clock
# DO NOT use set_input_delay -clock clk_in <delay value>
# [get_ports data_in]
set_input_delay -clock virt_clki <delay value> [get_ports data_in]
```

I/O Interface Uncertainty

Virtual clocks are recommended for I/O constraints because they most accurately represent the clocking topology of the design. An additional benefit is that you can specify different uncertainty values on clocks that interface with external I/O ports and clocks that feed register-to-register paths inside the FPGA.

Related Information

Clock Uncertainty on page 3-21
For more information about clock uncertainty and clock transfers.

Creating Generated Clocks

Define generated clocks on any nodes in your design which modify the properties of a clock signal, including phase, frequency, offset, and duty cycle. Generated clocks are most commonly used on the outputs of PLLs, on register clock dividers, clock muxes, and clocks forwarded to other devices from an FPGA output port, such as source synchronous and memory interfaces. In the SDC file, create generated clocks after the base clocks have been defined. Generated clocks automatically account for all clock delays and clock latency to the generated clock target.

Use the create_generated_clock command to constrain generated clocks in your design.

The -source option specifies the name of a node in the clock path which is used as reference for your generated clock. The source of the generated clock must be a node in your design netlist and not the name of a previously defined clock. You can use any node name on the clock path between the input clock pin of the target of the generated clock and the target node of its reference clock as the source node. A good practice is to specify the input clock pin of the target node as the source of your new generated clock. That
way, the source of the generated clock is decoupled from the naming and hierarchy of its clock source. If you change its clock source, you don’t have to edit the generated clock constraint.

If you have multiple base clocks feeding a node that is the source for a generated clock, you must define multiple generated clocks. Each generated clock is associated to one base clock using the -master_clock option in each generated clock statement. In some cases, generated clocks are generated with combinational logic. Depending on how your clock-modifying logic is synthesized, the name can change from compile to compile. If the name changes after you write the generated clock constraint, the generated clock is ignored because its target name no longer exists in the design. To avoid this problem, use a synthesis attribute or synthesis assignment to keep the final combinational node of the clock-modifying logic. Then use the kept name in your generated clock constraint. For details on keeping combinational nodes or wires, refer to the Implement as Output of Logic Cell logic option topic in Quartus Prime Help.

When a generated clock is created on a node that ultimately feeds the data input of a register, this creates a special case referred to as “clock-as-data”. Instances of clock-as-data are treated differently by TimeQuest. For example, when clock-as-data is used with DDR, both the rise and the fall of this clock need to be considered since it is a clock, and TimeQuest reports both rise and fall. With clock-as-data, the From Node is treated as the target of the generated clock, and the Launch Clock is treated as the generated clock. In the figure below, the first path is from toggle_clk (INVERTED) to clk, whereas the second path is from toggle_clk to clk. The slack in both cases is slightly different due to the difference in rise and fall times along the path; the ~5 ps difference can be seen in the Data Delay column. Only the path with the lowest slack value need be considered. This would also be true if this were not a clock-as-data case, but normally TimeQuest only reports the worst-case path between the two (rise and fall). In this example, if the generated clock were not defined on the register output, then only one path would be reported and it would be the one with the lowest slack value. If your design targets an Arria 10 device, the enhanced timing algorithms remove all common clock pessimism on paths treated as clock-as-data.

Figure 3-5: Example of clock-as-data

<table>
<thead>
<tr>
<th>Path #1: Setup slack is 9.166</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Summary</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Property</td>
</tr>
<tr>
<td>From Node</td>
</tr>
<tr>
<td>To Node</td>
</tr>
<tr>
<td>Launch Clock</td>
</tr>
<tr>
<td>Latch Clock</td>
</tr>
<tr>
<td>Relationship</td>
</tr>
<tr>
<td>Clock Skew</td>
</tr>
<tr>
<td>Data Delay</td>
</tr>
</tbody>
</table>

The TimeQuest analyzer provides the derive_pll_clocks command to automatically generate clocks for all PLL clock outputs. The properties of the generated clocks on the PLL outputs match the properties defined for the PLL.
Related Information

- **Deriving PLL Clocks** on page 3-18
  For more information about deriving PLL clock outputs.
- **Implement as Output of Logic Cell logic option**
  For more information on keeping combinational nodes or wires, refer to Quartus Prime Help.
- **create_generate_clock**
- **derive_pll_clocks**
- **create_generated_clocks**
  For information about these commands, refer to Quartus Prime Help.

Clock Divider Example

A common form of generated clock is a divide-by-two register clock divider. The following constraint creates a half-rate clock on the divide-by-two register.

**Figure 3-6: Clock Divider**

```plaintext
reg clk_sys
```

**Figure 3-7: Clock Divider Waveform**

<table>
<thead>
<tr>
<th>Edges</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_sys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk_div_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
cREATE_CLOCK -PERIOD 10 -NAME CLK_SYS [GET_PORTS CLK_SYS]
CREATE_GENERATED_CLOCK -NAME CLK_DIV_2 -DIVIDE_BY 2 -SOURCE [GET_PORTS CLK_SYS] [GET_PINS REG|Q]
```

Or in order to have the clock source be the clock pin of the register you can use:

```
cREATE_CLOCK -PERIOD 10 -NAME CLK_SYS [GET_PORTS CLK_SYS]
CREATE_GENERATED_CLOCK -NAME CLK_DIV_2 -DIVIDE_BY 2 -SOURCE [GET_PINS REG|CLK] [GET_PINS REG|Q]
```

Clock Multiplexor Example

Another common form of generated clock is on the output of a clock mux. One generated clock on the output is required for each input clock. The SDC example also includes the **set_clock_groups** command.
to indicate that the two generated clocks can never be active simultaneously in the design, so the
TimeQuest analyzer does not analyze cross-domain paths between the generated clocks on the output of
the clock mux.

Figure 3-8: Clock Mux

```plaintext
create_clock -name clock_a -period 10 [get_ports clk_a]
create_clock -name clock_b -period 10 [get_ports clk_b]
create_generated_clock -name clock_a_mux -source [get_ports clk_a] \
    [get_pins clk_mux|mux_out]
create_generated_clock -name clock_b_mux -source [get_ports clk_b] \
    [get_pins clk_mux|mux_out] -add
set_clock_groups -exclusive -group clock_a_mux -group clock_b_mux
```

### Deriving PLL Clocks

Use the `derive_pll_clocks` command to direct the TimeQuest analyzer to automatically search the
timing netlist for all unconstrained PLL output clocks. The `derive_pll_clocks` command detects your
current PLL settings and automatically creates generated clocks on the outputs of every PLL by calling the
`create_generated_clock` command.

### Create Base Clock for PLL input Clock Ports

```plaintext
create_clock -period 10.0 -name fpga_sys_clk [get_ports fpga_sys_clk] \
    derive_pll_clocks
```

If your design contains transceivers, LVDS transmitters, or LVDS receivers, you must use the
`derive_pll_clocks` command. The command automatically constrains this logic in your design and
creates timing exceptions for those blocks.

Include the `derive_pll_clocks` command in your SDC file after any `create_clock` command Each time
the TimeQuest analyzer reads your SDC file, the appropriate generate clock is created for each PLL output
clock pin. If a clock exists on a PLL output before running `derive_pll_clocks`, the pre-existing clock has
precedence, and an auto-generated clock is not created for that PLL output.

A simple PLL design with a register-to-register path.

Figure 3-9: Simple PLL Design

The TimeQuest analyzer generates messages when you use the `derive_pll_clocks` command to
automatically constrain the PLL for a design similar to the previous image.
Example 3-3: derive_pll_clocks Command Messages

Info:
Info: Deriving PLL Clocks:
Info: create_generated_clock -source pll_inst|altpll_component|pll|inclk[0] -
divide_by 2 -name
pll_inst|altpll_component|pll|clk[0] pll_inst|altpll_component|pll|clk[0]
Info:

The input clock pin of the PLL is the node pll_inst|altpll_component|pll|inclk[0] which is used
for the -source option. The name of the output clock of the PLL is the PLL output clock node, pll_inst|
altpll_component|pll|clk[0].

If the PLL is in clock switchover mode, multiple clocks are created for the output clock of the PLL; one
for the primary input clock (for example, inclk[0]), and one for the secondary input clock (for example,
inclk[1]). You should create exclusive clock groups for the primary and secondary output clocks since
they are not active simultaneously.

Related Information

- Creating Clock Groups on page 3-19
  For more information about creating exclusive clock groups.
- derive_pll_clocks
- Derive PLL Clocks
  For more information about the derive_pll_clocks command.

Creating Clock Groups

The TimeQuest analyzer assumes all clocks are related unless constrained otherwise.

To specify clocks in your design that are exclusive or asynchronous, use the set_clock_groups
command. The set_clock_groups command cuts timing between clocks in different groups, and
performs the same analysis regardless of whether you specify -exclusive or -asynchronous. A group is
defined with the -group option. The TimeQuest analyzer excludes the timing paths between clocks for
each of the separate groups.

The following tables show examples of various group options for the set_clock_groups command.

Table 3-1: set_clock_groups -group A

<table>
<thead>
<tr>
<th>Dest\Source</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Analyzed</td>
<td>Cut</td>
<td>Cut</td>
<td>Cut</td>
</tr>
<tr>
<td>B</td>
<td>Cut</td>
<td>Analyzed</td>
<td>Analyzed</td>
<td>Analyzed</td>
</tr>
<tr>
<td>C</td>
<td>Cut</td>
<td>Analyzed</td>
<td>Analyzed</td>
<td>Analyzed</td>
</tr>
<tr>
<td>D</td>
<td>Cut</td>
<td>Analyzed</td>
<td>Analyzed</td>
<td>Analyzed</td>
</tr>
</tbody>
</table>

Table 3-2: set_clock_groups -group {A B}

<table>
<thead>
<tr>
<th>Dest\Source</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Analyzed</td>
<td>Analyzed</td>
<td>Cut</td>
<td>Cut</td>
</tr>
</tbody>
</table>
Related Information

set_clock_groups
For more information about this command, refer to Quartus Prime Help.

Exclusive Clock Groups

Use the -exclusive option to declare that two clocks are mutually exclusive. You may want to declare clocks as mutually exclusive when multiple clocks are created on the same node. This case occurs for multiplexed clocks.

For example, an input port may be clocked by either a 25-MHz or a 50-MHz clock. To constrain this port, create two clocks on the port, and then create clock groups to declare that they do not coexist in the design at the same time. Declaring the clocks as mutually exclusive eliminates clock transfers that are derived between the 25-MHz clock and the 50-MHz clock.
Asynchronous Clock Groups

Use the \texttt{-asynchronous} option to create asynchronous clock groups. Asynchronous clock groups are commonly used to break the timing relationship where data is transferred through a FIFO between clocks running at different rates.

Related Information

\texttt{set_clock_groups}

For more information about this command, refer to Quartus Prime Help.

Accounting for Clock Effect Characteristics

The clocks you create with the TimeQuest analyzer are ideal clocks that do not account for any board effects. You can account for clock effect characteristics with clock latency and clock uncertainty.

Clock Latency

There are two forms of clock latency, clock source latency and clock network latency. Source latency is the propagation delay from the origin of the clock to the clock definition point (for example, a clock port). Network latency is the propagation delay from a clock definition point to a register’s clock pin. The total latency at a register’s clock pin is the sum of the source and network latencies in the clock path.

To specify source latency to any clock ports in your design, use the \texttt{set_clock_latency} command.

Note: The TimeQuest analyzer automatically computes network latencies; therefore, you only can characterize source latency with the \texttt{set_clock_latency} command. You must use the \texttt{-source} option.

Related Information

\texttt{set_clock_latency}

For more information about this command, refer to Quartus Prime Help.

Clock Uncertainty

When clocks are created, they are ideal and have perfect edges. It is important to add uncertainty to those perfect edges, to mimic clock-level effects like jitter. You should include the \texttt{derive_clock_uncertainty} command in your SDC file so that appropriate setup and hold uncertainties are automatically calculated and applied to all clock transfers in your design. If you don’t include the command, the TimeQuest analyzer performs it anyway; it is a critical part of constraining your design correctly.
The TimeQuest analyzer subtracts setup uncertainty from the data required time for each applicable path and adds the hold uncertainty to the data required time for each applicable path. This slightly reduces the setup and hold slack on each path.

The TimeQuest analyzer accounts for uncertainty clock effects for three types of clock-to-clock transfers; intraclock transfers, interclock transfers, and I/O interface clock transfers.

- Intraclock transfers occur when the register-to-register transfer takes place in the device and the source and destination clocks come from the same PLL output pin or clock port.
- Interclock transfers occur when a register-to-register transfer takes place in the core of the device and the source and destination clocks come from a different PLL output pin or clock port.
- I/O interface clock transfers occur when data transfers from an I/O port to the core of the device or from the core of the device to the I/O port.

To manually specify clock uncertainty, use the `set_clock_uncertainty` command. You can specify the uncertainty separately for setup and hold. You can also specify separate values for rising and falling clock transitions, although this is not commonly used. You can override the value that was automatically applied by the `derive_clock_uncertainty` command, or you can add to it.

The `derive_clock_uncertainty` command accounts for PLL clock jitter if the clock jitter on the input to a PLL is within the input jitter specification for PLL's in the specified device. If the input clock jitter for the PLL exceeds the specification, you should add additional uncertainty to your PLL output clocks to account for excess jitter with the `set_clock_uncertainty -add` command. Refer to the device handbook for your device for jitter specifications.

Another example is to use `set_clock_uncertainty -add` to add uncertainty to account for peak-to-peak jitter from a board when the jitter exceeds the jitter specification for that device. In this case you would add uncertainty to both setup and hold equal to 1/2 the jitter value:

```plaintext
set_clock_uncertainty -setup -to <clock name> \  
   -setup -add <p2p jitter/2>

set_clock_uncertainty -hold -enable_same_physical_edge -to <clock name> \ 
   -add <p2p jitter/2>
```

There is a complex set of precedence rules for how the TimeQuest analyzer applies values from `derive_clock_uncertainty` and `set_clock_uncertainty`, which depend on the order the commands appear in your SDC files, and various options used with the commands. The Help topics referred to below contain complete descriptions of these rules. These precedence rules are much simpler to understand and implement if you follow these recommendations:

- If you want to assign your own clock uncertainty values to any clock transfers, the best practice is to put your `set_clock_uncertainty` exceptions after the `derive_clock_uncertainty` command in your SDC file.
- When you use the `-add` option for `set_clock_uncertainty`, the value you specify is added to the value from `derive_clock_uncertainty`. If you don’t specify `-add`, the value you specify replaces the value from `derive_clock_uncertainty`.

Related Information

- `set_clock_uncertainty`
- `derive_clock_uncertainty`
- `remove_clock_uncertainty`

For more information about these commands, refer to Quartus Prime Help.
Creating I/O Requirements

The TimeQuest analyzer reviews setup and hold relationships for designs in which an external source interacts with a register internal to the design. The TimeQuest analyzer supports input and output external delay modeling with the `set_input_delay` and `set_output_delay` commands. You can specify the clock and minimum and maximum arrival times relative to the clock.

You must specify timing requirements, including internal and external timing requirements, before you fully analyze a design. With external timing requirements specified, the TimeQuest analyzer verifies the I/O interface, or periphery of the device, against any system specification.

Input Constraints

Input constraints allow you to specify all the external delays feeding into the device. Specify input requirements for all input ports in your design.

You can use the `set_input_delay` command to specify external input delay requirements. Use the `-clock` option to reference a virtual clock. Using a virtual clock allows the TimeQuest analyzer to correctly derive clock uncertainties for interclock and intraclock transfers. The virtual clock defines the launching clock for the input port. The TimeQuest analyzer automatically determines the latching clock inside the device that captures the input data, because all clocks in the device are defined.

**Figure 3-11: Input Delay**

![Input Delay Diagram]

The calculation the TimeQuest analyzer performs to determine the typical input delay.

**Figure 3-12: Input Delay Calculation**

\[
\text{input delay}_{\text{MAX}} = (\text{cd}_{\text{ext}} - \text{cd}_{\text{altr}}) + \text{tco}_{\text{ext}} + \text{dd}\]
\[
\text{input delay}_{\text{MIN}} = (\text{cd}_{\text{ext}} - \text{cd}_{\text{altr}}) + \text{tco}_{\text{ext}} + \text{dd}
\]

Output Constraints

Output constraints allow you to specify all external delays from the device for all output ports in your design.

You can use the `set_output_delay` command to specify external output delay requirements. Use the `-clock` option to reference a virtual clock. The virtual clock defines the latching clock for the output port. The TimeQuest analyzer automatically determines the launching clock inside the device that launches the output data, because all clocks in the device are defined. The following figure is an example of an output delay referencing a virtual clock.
The calculation the TimeQuest analyzer performs to determine the typical output delay.

**Figure 3-14: Output Delay Calculation**

\[
\begin{align*}
\text{output delay}_{\text{MAX}} &= \text{dd}_{\text{MAX}} + \text{tsu}_{\text{ext}} + (\text{cd}_{\text{altr}}_{\text{MAX}} - \text{cd}_{\text{ext}}_{\text{MIN}}) \\
\text{output delay}_{\text{MIN}} &= (\text{dd}_{\text{MIN}} - \text{th}_{\text{ext}} + (\text{cd}_{\text{altr}}_{\text{MIN}} - \text{cd}_{\text{ext}}_{\text{MAX}}))
\end{align*}
\]

**Related Information**
- `set_input_delay`
- `set_output_delay`
  
  For more information about these commands, refer to Quartus Prime Help.

**Creating Delay and Skew Constraints**

The TimeQuest analyzer supports the Synopsys Design Constraint format for constraining timing for the ports in your design. These constraints allow the TimeQuest analyzer to perform a system static timing analysis that includes not only the device internal timing, but also any external device timing and board timing parameters.

**Advanced I/O Timing and Board Trace Model Delay**

The TimeQuest analyzer can use advanced I/O timing and board trace model assignments to model I/O buffer delays in your design.

If you change any advanced I/O timing settings or board trace model assignments, recompile your design before you analyze timing, or use the `-force_dat` option to force delay annotation when you create a timing netlist.

**Example 3-4: Forcing Delay Annotation**

```
create_timing_netlist -force_dat
```

**Related Information**
- Using Advanced I/O Timing
- I/O Management
  
  For more information about advanced I/O timing.
Maximum Skew

To specify the maximum path-based skew requirements for registers and ports in the design and report the results of maximum skew analysis, use the set_max_skew command in conjunction with the report_max_skew command.

Use the set_max_skew constraint to perform maximum allowable skew analysis between sets of registers or ports. In order to constrain skew across multiple paths, all such paths must be defined within a single set_max_skew constraint. set_max_skew timing constraint is not affected by set_max_delay, set_min_delay, and set_multicycle_path but it does obey set_false_path and set_clock_groups. If your design targets an Arria 10 device, skew constraints are not affected by set_clock_groups.

Table 3-6: set_max_skew Options

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>-help</td>
</tr>
<tr>
<td>-long_help</td>
<td>Long help with examples and possible return values</td>
</tr>
<tr>
<td>-exclude &lt;Tcl list&gt;</td>
<td>A Tcl list of parameters to exclude during skew analysis. This list can include one or more of the following: utsu, uth, utco, from_clock, to_clock, clock_uncertainty, ccpp, input_delay, output_delay, odv.</td>
</tr>
<tr>
<td>Note: Not supported for Arria 10 devices.</td>
<td></td>
</tr>
<tr>
<td>-fall_from_clock &lt;names&gt;</td>
<td>Valid source clocks (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td>-fall_to_clock &lt;names&gt;</td>
<td>Valid destination clocks (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td>-from &lt;names&gt;(1)</td>
<td>Valid sources (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td>-from_clock &lt;names&gt;</td>
<td>Valid source clocks (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td>-get_skew_value_from_clock_period &lt;src_clock_period</td>
<td>dst_clock_period</td>
</tr>
<tr>
<td>-include &lt;Tcl list&gt;</td>
<td>Tcl list of parameters to include during skew analysis. This list can include one or more of the following: utsu, uth, utco, from_clock, to_clock, clock_uncertainty, ccpp, input_delay, output_delay, odv.</td>
</tr>
<tr>
<td>Note: Not supported for Arria 10 devices.</td>
<td></td>
</tr>
<tr>
<td>-rise_from_clock &lt;names&gt;</td>
<td>Valid source clocks (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td>-rise_to_clock &lt;names&gt;</td>
<td>Valid destination clocks (string patterns are matched using Tcl string matching)</td>
</tr>
</tbody>
</table>

(1) Legal values for the -from and -to options are collections of clocks, registers, ports, pins, cells or partitions in a design.
### Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-skew_value_multiplier</code></td>
<td>Value by which the clock period should be multiplied to compute skew requirement.</td>
</tr>
<tr>
<td><code>&lt;multiplier&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>-to &lt;names&gt;</code></td>
<td>Valid destinations (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td><code>-to_clock &lt;names&gt;</code></td>
<td>Valid destination clocks (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td><code>&lt;skew&gt;</code></td>
<td>Required skew</td>
</tr>
</tbody>
</table>

Applying maximum skew constraints between clocks applies the constraint from all register or ports driven by the clock specified with the `-from` option to all registers or ports driven by the clock specified with the `-to` option.

Use the `-include` and `-exclude` options to include or exclude one or more of the following: register micro parameters (`utsu, uth, utco`), clock arrival times (`from_clock, to_clock`), clock uncertainty (`clock_uncertainty`), common clock path pessimism removal (`ccpp`), input and output delays (`input_delay, output_delay`) and on-die variation (`odv`). Max skew analysis can include data arrival times, clock arrival times, register micro parameters, clock uncertainty, on-die variation and ccpp removal. Among these, only ccpp removal is disabled during the Fitter by default. When `-include` is used, those in the inclusion list are added to the default analysis. Similarly, when `-exclude` is used, those in the exclusion list are excluded from the default analysis. When both the `-include` and `-exclude` options specify the same parameter, that parameter is excluded.

**Note:** If your design targets an Arria 10 device, `-exclude` and `-include` are not supported.

Use `-get_skew_value_from_clock_period` to set the skew as a multiple of the launching or latching clock period, or whichever of the two has a smaller period. If this option is used, then `-skew_value_multiplier` must also be set, and the positional skew option may not be set. If the set of skew paths is clocked by more than one clock, TimeQuest uses the one with smallest period to compute the skew constraint.

When this constraint is used, results of max skew analysis are displayed in the Report Max Skew (`report_max_skew`) report from the TimeQuest Timing Analyzer. Since skew is defined between two or more paths, no results are displayed if the `-from/-from_clock` and `-to/-to_clock` filters satisfy less than two paths.

**Related Information**

- `set_max_skew`
- `report_max_skew`

For more information about these commands, refer to Quartus Prime Help.

### Net Delay

Use the `set_net_delay` command to set the net delays and perform minimum or maximum timing analysis across nets. The `-from` and `-to` options can be string patterns or pin, port, register, or net collections. When pin or net collection is used, the collection should include output pins or nets.
Table 3-7: set_net_delay Options

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`-h</td>
<td>-help`</td>
</tr>
<tr>
<td><code>-long_help</code></td>
<td>Long help with examples and possible return values</td>
</tr>
<tr>
<td><code>-from &lt;names&gt;</code></td>
<td>Valid source pins, ports, registers or nets (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td><code>-get_value_from_clock_period</code></td>
<td>Option to interpret net delay constraint as a multiple of the clock period.</td>
</tr>
<tr>
<td>`&lt;src_clock_period</td>
<td>dst_clock_period</td>
</tr>
<tr>
<td><code>-max</code></td>
<td>Specifies maximum delay</td>
</tr>
<tr>
<td><code>-min</code></td>
<td>Specifies minimum delay</td>
</tr>
<tr>
<td><code>-to &lt;names&gt;</code>(2)</td>
<td>Valid destination pins, ports, registers or nets (string patterns are matched using Tcl string matching)</td>
</tr>
<tr>
<td><code>-value_multiplier &lt;multiplier&gt;</code></td>
<td>Value by which the clock period should be multiplied to compute net delay requirement.</td>
</tr>
<tr>
<td><code>&lt;delay&gt;</code></td>
<td>Required delay</td>
</tr>
</tbody>
</table>

When you use the `-min` option, slack is calculated by looking at the minimum delay on the edge. If you use `-max` option, slack is calculated with the maximum edge delay.

Use `-get_skew_value_from_clock_period` to set the net delay requirement as a multiple of the launching or latching clock period, or whichever of the two has a smaller or larger period. If this option is used, then `-value_multiplier` must also be set, and the positional delay option may not be set. If the set of nets is clocked by more than one clock, TimeQuest uses the net with smallest period to compute the constraint for a `-max` constraint, and the largest period for a `-min` constraint. If there are no clocks clocking the endpoints of the net (e.g. if the endpoints of the nets are not registers or constraint ports), then the net delay constraint will be ignored.

Related Information
- `set_net_delay`
- `report_net_delay`

For more information about these commands, refer to Quartus Prime Help.

Using create_timing_netlist

You can configure or load the timing netlist that the TimeQuest analyzer uses to calculate path delay data.

Your design should have a timing netlist before running the TimeQuest analyzer. You can use the Create Timing Netlist dialog box or the Create Timing Netlist command in the Tasks pane. The command also generates Advanced I/O Timing reports if you turned on Enable Advanced I/O Timing in the TimeQuest Timing Analyzer page of the Settings dialog box.

Note: The timing netlist created is based on the initial configuration of the design. Any configuration changes done by the design after the device enters user mode, for example, dynamic transceiver

---

(2) If the `-to` option is unused or if the `-to` filter is a wildcard ("*") character, all the output pins and registers on timing netlist became valid destination points.
reconfiguration, are not reflected in the timing netlist. This applies to all device families except transceivers on Arria 10 devices with the Multiple Reconfiguration Profiles feature.

The following diagram shows how the TimeQuest analyzer interprets and classifies timing netlist data for a sample design.

**Figure 3-15: How TimeQuest Interprets the Timing Netlist**

---

### Creating Timing Exceptions

Timing exceptions in the TimeQuest analyzer provide a way to modify the default timing analysis behavior to match the analysis required by your design. Specify timing exceptions after clocks and input and output delay constraints because timing exceptions can modify the default analysis.

#### Precedence

If a conflict of node names occurs between timing exceptions, the following order of precedence applies:

1. False path
2. Minimum delays and maximum delays
3. Multicycle path

The false path timing exception has the highest precedence. Within each category, assignments to individual nodes have precedence over assignments to clocks. Finally, the remaining precedence for additional conflicts is order-dependent, such that the assignments most recently created overwrite, or partially overwrite, earlier assignments.

#### False Paths

Specifying a false path in your design removes the path from timing analysis.

Use the `set_false_path` command to specify false paths in your design. You can specify either a point-to-point or clock-to-clock path as a false path. For example, a path you should specify as false path is a static configuration register that is written once during power-up initialization, but does not change state again. Although signals from static configuration registers often cross clock domains, you may not want to make false path exceptions to a clock-to-clock path, because some data may transfer across clock domains. However, you can selectively make false path exceptions from the static configuration register to all endpoints.
To make false path exceptions from all registers beginning with A to all registers beginning with B, use the following code in your SDC file.

```
set_false_path -from [get_pins A*] -to [get_pins B*]
```

The TimeQuest analyzer assumes all clocks are related unless you specify otherwise. Clock groups are a more efficient way to make false path exceptions between clocks, compared to writing multiple `set_false_path` exceptions between every clock transfer you want to eliminate.

Related Information
- **Creating Clock Groups** on page 3-19
  - For more information about creating exclusive clock groups.
- **set_false_path**
  - For more information about this command, refer to Quartus Prime Help.

**Minimum and Maximum Delays**

To specify an absolute minimum or maximum delay for a path, use the `set_min_delay` command or the `set_max_delay` commands, respectively. Specifying minimum and maximum delay directly overwrites existing setup and hold relationships with the minimum and maximum values.

Use the `set_max_delay` and `set_min_delay` commands to create constraints for asynchronous signals that do not have a specific clock relationship in your design, but require a minimum and maximum path delay. You can create minimum and maximum delay exceptions for port-to-port paths through the device without a register stage in the path. If you use minimum and maximum delay exceptions to constrain the path delay, specify both the minimum and maximum delay of the path; do not constrain only the minimum or maximum value.

If the source or destination node is clocked, the TimeQuest analyzer takes into account the clock paths, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the minimum or maximum delay check.

If you specify a minimum or maximum delay between timing nodes, the delay applies only to the path between the two nodes. If you specify a minimum or maximum delay for a clock, the delay applies to all paths where the source node or destination node is clocked by the clock.

You can create a minimum or maximum delay exception for an output port that does not have an output delay constraint. You cannot report timing for the paths associated with the output port; however, the TimeQuest analyzer reports any slack for the path in the setup summary and hold summary reports. Because there is no clock associated with the output port, no clock is reported for timing paths associated with the output port.

**Note:** To report timing with clock filters for output paths with minimum and maximum delay constraints, you can set the output delay for the output port with a value of zero. You can use an existing clock from the design or a virtual clock as the clock reference.

Related Information
- **set_max_delay**
- **set_min_delay**
  - For more information about these commands, refer to Quartus Prime Help.

**Delay Annotation**

To modify the default delay values used during timing analysis, use the `set_annotated_delay` and `set_timing_derate` commands. You must update the timing netlist to see the results of these commands.
To specify different operating conditions in a single SDC file, rather than having multiple SDC files that specify different operating conditions, use the `set_annotated_delay -operating_conditions` command.

Related Information

- `set_timing_derate`
- `set_annotated_delay`

For more information about these commands, refer to the Quartus Prime Help.

**Multicycle Paths**

By default, the TimeQuest analyzer performs a single-cycle analysis, which is the most restrictive type of analysis. When analyzing a path, the setup launch and latch edge times are determined by finding the closest two active edges in the respective waveforms.

For a hold analysis, the timing analyzer analyzes the path against two timing conditions for every possible setup relationship, not just the worst-case setup relationship. Therefore, the hold launch and latch times may be completely unrelated to the setup launch and latch edges. The TimeQuest analyzer does not report negative setup or hold relationships. When either a negative setup or a negative hold relationship is calculated, the TimeQuest analyzer moves both the launch and latch edges such that the setup and hold relationship becomes positive.

A multicycle constraint adjusts setup or hold relationships by the specified number of clock cycles based on the source (`-start`) or destination (`-end`) clock. An end setup multicycle constraint of 2 extends the worst-case setup latch edge by one destination clock period. If `-start` and `-end` values are not specified, the default constraint is `-end`.

Hold multicycle constraints are based on the default hold position (the default value is 0). An end hold multicycle constraint of 1 effectively subtracts one destination clock period from the default hold latch edge.

When the objects are timing nodes, the multicycle constraint only applies to the path between the two nodes. When an object is a clock, the multicycle constraint applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock. When you adjust a setup relationship with a multicycle constraint, the hold relationship is adjusted automatically.

You can use TimeQuest analyzer commands to modify either the launch or latch edge times that the uses to determine a setup relationship or hold relationship.

**Table 3-8: Commands to Modify Edge Times**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description of Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_multicycle_path -setup -end &lt;value&gt;</code></td>
<td>Latch edge time of the setup relationship</td>
</tr>
<tr>
<td><code>set_multicycle_path -setup -start&lt;value&gt;</code></td>
<td>Launch edge time of the setup relationship</td>
</tr>
<tr>
<td><code>set_multicycle_path -hold -end &lt;value&gt;</code></td>
<td>Latch edge time of the hold relationship</td>
</tr>
<tr>
<td><code>set_multicycle_path -hold -start &lt;value&gt;</code></td>
<td>Launch edge time of the hold relationship</td>
</tr>
</tbody>
</table>

**Common Multicycle Variations**

Multicycle exceptions adjust the timing requirements for a register-to-register path, allowing the Fitter to optimally place and route a design in a device. Multicycle exceptions also can reduce compilation time and improve the quality of results, and can be used to change timing requirements. Two common
multicycle variations are relaxing setup to allow a slower data transfer rate, and altering the setup to account for a phase shift.

**Relaxing Setup with set_multicycle_path**

A common type of multicycle exception occurs when the data transfer rate is slower than the clock cycle. Relaxing the setup relationship opens the window when data is accepted as valid.

In this example, the source clock has a period of 10 ns, but a group of registers are enabled by a toggling clock, so they only toggle every other cycle. Since they are fed by a 10 ns clock, the TimeQuest analyzer reports a setup of 10 ns and a hold of 0 ns. However, since the data is transferring every other cycle, the relationships should be analyzed as if the clock were operating at 20 ns, which would result in a setup of 20 ns, while the hold remains 0 ns, in essence, extending the window of time when the data can be recognized.

The following pair of multicycle assignments relax the setup relationship by specifying the `-setup` value of N and the `-hold` value as N-1. You must specify the hold relationship with a `-hold` assignment to prevent a positive hold requirement.

**Relaxing Setup while Maintaining Hold**

```
set_multicycle_path -setup -from src_reg* -to dst_reg* 2
set_multicycle_path -hold -from src_reg* -to dst_reg* 1
```

**Figure 3-16: Relaxing Setup by Multiple Cycles**

This pattern can be extended to create larger setup relationships in order to ease timing closure requirements. A common use for this exception is when writing to asynchronous RAM across an I/O interface. The delay between address, data, and a write enable may be several cycles. A multicycle exception to I/O ports can allow extra time for the address and data to resolve before the enable occurs.

You can relax the setup by three cycles with the following code in your SDC file.
Three Cycle I/O Interface Exception

```plaintext
set_multicycle_path -setup -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]}] 3
set_multicycle_path -hold -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]}] 2
```

Accounting for a Phase Shift

In this example, the design contains a PLL that performs a phase-shift on a clock whose domain exchanges data with domains that do not experience the phase shift. A common example is when the destination clock is phase-shifted forward and the source clock is not, the default setup relationship becomes that phase-shift, thus shifting the window when data is accepted as valid.

For example, the following code is a circumstance where a PLL phase-shifts one output forward by a small amount, in this case 0.2 ns.

Cross Domain Phase-Shift

```plaintext
create_generated_clock -source pll|inclk[0] -name pll|clk[0] pll|clk[0]
```

The default setup relationship for this phase-shift is 0.2 ns, shown in Figure A, creating a scenario where the hold relationship is negative, which makes achieving timing closure nearly impossible.

**Figure 3-17: Phase-Shifted Setup and Hold**

![Figure 3-17](image)

Adding the following constraint in your SDC allows the data to transfer to the following edge.

```plaintext
set_multicycle_path -setup -from [get_clocks clk_a] -to [get_clocks clk_b] 2
```

The hold relationship is derived from the setup relationship, making a multicyle hold constraint unnecessary.

Related Information

- **Same Frequency Clocks with Destination Clock Offset** on page 3-41
  Refer to this topic for a more complete example.
- **Same Frequency Clocks with Destination Clock Offset** on page 3-41
  Refer to this topic for a more complete example.
- **set_multicycle_path**
  For more information about this command, refer to the Quartus Prime Help.
Examples of Basic Multicycle Exceptions

Each example explains how the multicycle exceptions affect the default setup and hold analysis in the TimeQuest analyzer. The multicycle exceptions are applied to a simple register-to-register circuit. Both the source and destination clocks are set to 10 ns.

Default Settings

By default, the TimeQuest analyzer performs a single-cycle analysis to determine the setup and hold checks. Also, by default, the TimeQuest analyzer sets the end multicycle setup assignment value to one and the end multicycle hold assignment value to zero.

The source and the destination timing waveform for the source register and destination register, respectively where HC1 and HC2 are hold checks one and two and SC is the setup check.

Figure 3-18: Default Timing Diagram

![Default Timing Diagram](image)

The calculation that the TimeQuest analyzer performs to determine the setup check.

**Figure 3-19: Setup Check**

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 10 \text{ ns} - 0 \text{ ns} \\
= 10 \text{ ns}
\]

The most restrictive setup relationship with the default single-cycle analysis, that is, a setup relationship with an end multicycle setup assignment of one, is 10 ns.

The setup report for the default setup in the TimeQuest analyzer with the launch and latch edges highlighted.
The calculation that the TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

**Figure 3-21: Hold Check**

\[
\begin{align*}
\text{hold check 1} & \quad = \quad \text{current launch edge} - \text{previous latch edge} \\
& \quad = \quad 0 \text{ ns} - 0 \text{ ns} \\
& \quad = \quad 0 \text{ ns} \\
\text{hold check 2} & \quad = \quad \text{next launch edge} - \text{current latch edge} \\
& \quad = \quad 10 \text{ ns} - 10 \text{ ns} \\
& \quad = \quad 0 \text{ ns}
\end{align*}
\]

The most restrictive hold relationship with the default single-cycle analysis, that a hold relationship with an end multicycle hold assignment of zero, is 0 ns.

The hold report for the default setup in the TimeQuest analyzer with the launch and latch edges highlighted.
End Multicycle Setup = 2 and End Multicycle Hold = 0

In this example, the end multicycle setup assignment value is two, and the end multicycle hold assignment value is zero.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
-setup -end 2
```

**Note:** An end multicycle hold value is not required because the default end multicycle hold value is zero. In this example, the setup relationship is relaxed by a full clock period by moving the latch edge to the next latch edge. The hold analysis is unchanged from the default settings.

The setup timing diagram for the analysis that the TimeQuest analyzer performs. The latch edge is a clock cycle later than in the default single-cycle analysis.
The calculation that the TimeQuest analyzer performs to determine the setup check.

**Figure 3-24: Setup Check**

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 20 \text{ ns} - 0 \text{ ns} \\
= 20 \text{ ns}
\]

The most restrictive setup relationship with an end multicycle setup assignment of two is 20 ns.

The setup report in the TimeQuest analyzer with the launch and latch edges highlighted.

**Figure 3-25: Setup Report**
Because the multicycle hold latch and launch edges are the same as the results of hold analysis with the default settings, the multicycle hold analysis in this example is equivalent to the single-cycle hold analysis. The hold checks are relative to the setup check. Usually, the TimeQuest analyzer performs hold checks on every possible setup check, not only on the most restrictive setup check edges.

**Figure 3-26: Hold Timing Diagram**

![Hold Timing Diagram](image)

The calculation that the TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

**Figure 3-27:**

\[
\text{hold check 1} = \text{current launch edge} - \text{previous latch edge} \\
= 0 \text{ ns} - 10 \text{ ns} \\
= -10 \text{ ns}
\]

\[
\text{hold check 2} = \text{next launch edge} - \text{current latch edge} \\
= 10 \text{ ns} - 20 \text{ ns} \\
= -10 \text{ ns}
\]

The most restrictive hold relationship with an end multicycle setup assignment value of two and an end multicycle hold assignment value of zero is 10 ns.

The hold report for this example in the TimeQuest analyzer with the launch and latch edges highlighted.
**End Multicycle Setup = 2 and End Multicycle Hold = 1**

In this example, the end multicycle setup assignment value is two, and the end multicycle hold assignment value is one.

**Multicycle Exceptions**

```bash
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] -setup -end 2
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] -hold -end 1
```

In this example, the setup relationship is relaxed by two clock periods by moving the latch edge to the left two clock periods. The hold relationship is relaxed by a full period by moving the latch edge to the previous latch edge.

The setup timing diagram for the analysis that the TimeQuest analyzer performs.
The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-30: Setup Check

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 20 \text{ ns} - 0 \text{ ns} \\
= 20 \text{ ns}
\]

The most restrictive hold relationship with an end multicycle setup assignment value of two is 20 ns.

The setup report for this example in the TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-31: Setup Report
The timing diagram for the hold checks for this example. The hold checks are relative to the setup check.

**Figure 3-32: Hold Timing Diagram**

The calculation that the TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

**Figure 3-33: Hold Check**

\[
\text{hold check 1} = \text{current launch edge} - \text{previous latch edge} \\
= 0 \text{ ns} - 0 \text{ ns} \\
= 0 \text{ ns}
\]

\[
\text{hold check 2} = \text{next launch edge} - \text{current latch edge} \\
= 10 \text{ ns} - 10 \text{ ns} \\
= 0 \text{ ns}
\]

The most restrictive hold relationship with an end multicycle setup assignment value of two and an end multicycle hold assignment value of one is 0 ns.

The hold report for this example in the TimeQuest analyzer with the launch and latch edges highlighted.
Application of Multicycle Exceptions

This section shows the following examples of applications of multicycle exceptions. Each example explains how the multicycle exceptions affect the default setup and hold analysis in the TimeQuest analyzer. All of the examples are between related clock domains. If your design contains related clocks, such as PLL clocks, and paths between related clock domains, you can apply multicycle constraints.

Same Frequency Clocks with Destination Clock Offset

In this example, the source and destination clocks have the same frequency, but the destination clock is offset with a positive phase shift. Both the source and destination clocks have a period of 10 ns. The destination clock has a positive phase shift of 2 ns with respect to the source clock.

An example of a design with same frequency clocks and a destination clock offset.

Figure 3-35: Same Frequency Clocks with Destination Clock Offset

The timing diagram for default setup check analysis that the TimeQuest analyzer performs.
The calculation that the TimeQuest analyzer performs to determine the setup check.

**Figure 3-37: Setup Check**

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 2 \text{ ns} - 0 \text{ ns} \\
= 2 \text{ ns}
\]

The setup relationship shown is too pessimistic and is not the setup relationship required for typical designs. To correct the default analysis, you must use an end multicycle setup exception of two. A multicycle exception used to correct the default analysis in this example in your SDC file.

**Multicycle Exceptions**

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \\
    -setup -end 2
```

The timing diagram for the preferred setup relationship for this example.

**Figure 3-38: Preferred Setup Relationship**
The timing diagram for default hold check analysis that the TimeQuest analyzer performs with an end multicycle setup value of two.

**Figure 3-39: Default Hold Check**

![Timing Diagram]

The calculation that the TimeQuest analyzer performs to determine the hold check.

**Figure 3-40: Hold Check**

\[
\text{hold check 1} = \text{current launch edge} - \text{previous latch edge} \\
= 0 \text{ ns} - 2 \text{ ns} \\
= -2 \text{ ns}
\]

\[
\text{hold check 2} = \text{next launch edge} - \text{current latch edge} \\
= 10 \text{ ns} - 12 \text{ ns} \\
= -2 \text{ ns}
\]

In this example, the default hold analysis returns the preferred hold requirements and no multicycle hold exceptions are required.

The associated setup and hold analysis if the phase shift is –2 ns. In this example, the default hold analysis is correct for the negative phase shift of 2 ns, and no multicycle exceptions are required.
Destination Clock Frequency is a Multiple of the Source Clock Frequency

In this example, the destination clock frequency value of 5 ns is an integer multiple of the source clock frequency of 10 ns. The destination clock frequency can be an integer multiple of the source clock frequency when a PLL is used to generate both clocks with a phase shift applied to the destination clock.

An example of a design where the destination clock frequency is a multiple of the source clock frequency.

The timing diagram for default setup check analysis that the TimeQuest analyzer performs.

The calculation that the TimeQuest analyzer performs to determine the setup check.
Setup Check

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge}
\]

\[
= 5 \text{ ns} - 0 \text{ ns}
\]

\[
= 5 \text{ ns}
\]

The setup relationship demonstrates that the data does not need to be captured at edge one, but can be captured at edge two; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the latch edge by one clock period with an end multicycle setup exception of two. The multicycle exception assignment used to correct the default analysis in this example.

**Multicycle Exceptions**

```plaintext
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \\
-setup -end 2
```

The timing diagram for the preferred setup relationship for this example.

**Figure 3-45: Preferred Setup Analysis**

The timing diagram for default hold check analysis performed by the TimeQuest analyzer with an end multicycle setup value of two.

**Figure 3-46: Default Hold Check**
The calculation that the TimeQuest analyzer performs to determine the hold check.

**Figure 3-47: Hold Check**

```
hold check 1 = current launch edge – previous latch edge
    = 0 ns – 5 ns
    = –5 ns

hold check 2 = next launch edge – current latch edge
    = 10 ns – 10 ns
    = 0 ns
```

In this example, hold check one is too restrictive. The data is launched by the edge at 0 ns and should check against the data captured by the previous latch edge at 0 ns, which does not occur in hold check one. To correct the default analysis, you must use an end multicycle hold exception of one.

**Destination Clock Frequency is a Multiple of the Source Clock Frequency with an Offset**

This example is a combination of the previous two examples. The destination clock frequency is an integer multiple of the source clock frequency and the destination clock has a positive phase shift. The destination clock frequency is 5 ns and the source clock frequency is 10 ns. The destination clock also has a positive offset of 2 ns with respect to the source clock. The destination clock frequency can be an integer multiple of the source clock frequency with an offset when a PLL is used to generate both clocks with a phase shift applied to the destination clock. The following example shows a design in which the destination clock frequency is a multiple of the source clock frequency with an offset.

**Figure 3-48: Destination Clock is Multiple of Source Clock with Offset**

The timing diagram for the default setup check analysis the TimeQuest analyzer performs.
The calculation that the TimeQuest analyzer performs to determine the setup check.

**Figure 3-50: Hold Check**

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} = 2 \text{ ns} - 0 \text{ ns} = 2 \text{ ns}
\]

The setup relationship in this example demonstrates that the data does not need to be captured at edge one, but can be captured at edge two; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the latch edge by one clock period with an end multicycle setup exception of three.

The multicycle exception code you can use to correct the default analysis in this example.

**Multicycle Exceptions**

\[
\text{set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \-setup -end 3}
\]

The timing diagram for the preferred setup relationship for this example.

**Figure 3-51: Preferred Setup Analysis**
The timing diagram for default hold check analysis the TimeQuest analyzer performs with an end multicycle setup value of three.

**Figure 3-52: Default Hold Check**

![Timing Diagram](image)

The calculation that the TimeQuest analyzer performs to determine the hold check.

**Figure 3-53: Hold Check**

\[
\begin{align*}
\text{hold check 1} & = \text{current launch edge} - \text{previous latch edge} \\
& = 0 \text{ ns} - 5 \text{ ns} \\
& = -5 \text{ ns} \\
\text{hold check 2} & = \text{next launch edge} - \text{current latch edge} \\
& = 10 \text{ ns} - 10 \text{ ns} \\
& = 0 \text{ ns}
\end{align*}
\]

In this example, hold check one is too restrictive. The data is launched by the edge at 0 ns and should check against the data captured by the previous latch edge at 2 ns, which does not occur in hold check one. To correct the default analysis, you must use an end multicycle hold exception of one.

**Source Clock Frequency is a Multiple of the Destination Clock Frequency**

In this example, the source clock frequency value of 5 ns is an integer multiple of the destination clock frequency of 10 ns. The source clock frequency can be an integer multiple of the destination clock frequency when a PLL is used to generate both clocks and different multiplication and division factors are used.

An example of a design where the source clock frequency is a multiple of the destination clock frequency.

**Figure 3-54: Source Clock Frequency is Multiple of Destination Clock Frequency**

![Circuit Diagram](image)
The timing diagram for default setup check analysis performed by the TimeQuest analyzer.

**Figure 3-55: Default Setup Check Analysis**

The calculation that the TimeQuest analyzer performs to determine the setup check.

**Figure 3-56: Setup Check**

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 10 \text{ ns} - 5 \text{ ns} \\
= 5 \text{ ns}
\]

The setup relationship shown demonstrates that the data launched at edge one does not need to be captured, and the data launched at edge two must be captured; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the launch edge by one clock period with a start multicycle setup exception of two.

The multicycle exception code you can use to correct the default analysis in this example.

**Multicycle Exceptions**

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \\
    -setup -start 2
```

The timing diagram for the preferred setup relationship for this example.
The timing diagram for default hold check analysis the TimeQuest analyzer performs for a start multicycle setup value of two.

**Figure 3-58: Default Hold Check**

The calculation that the TimeQuest analyzer performs to determine the hold check.

**Figure 3-59: Hold Check**

\[
\text{hold check 1} = \text{current launch edge} - \text{previous latch edge} \\
= 0 \text{ ns} - 0 \text{ ns} \\
= 0 \text{ ns}
\]

\[
\text{hold check 2} = \text{next launch edge} - \text{current latch edge} \\
= 5 \text{ ns} - 10 \text{ ns} \\
= -5 \text{ ns}
\]

In this example, hold check two is too restrictive. The data is launched next by the edge at 10 ns and should check against the data captured by the current latch edge at 10 ns, which does not occur in hold check two. To correct the default analysis, you must use a start multicycle hold exception of one.
Source Clock Frequency is a Multiple of the Destination Clock Frequency with an Offset

In this example, the source clock frequency is an integer multiple of the destination clock frequency and the destination clock has a positive phase offset. The source clock frequency is 5 ns and destination clock frequency is 10 ns. The destination clock also has a positive offset of 2 ns with respect to the source clock. The source clock frequency can be an integer multiple of the destination clock frequency with an offset when a PLL is used to generate both clocks, different multiplication.

Figure 3-60: Source Clock Frequency is Multiple of Destination Clock Frequency with Offset

Timing diagram for default setup check analysis the TimeQuest analyzer performs.

Figure 3-61: Setup Timing Diagram

The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-62: Setup Check

\[
\text{setup check} = \text{current latch edge} - \text{closest previous launch edge} \\
= 12 \text{ ns} - 10 \text{ ns} \\
= 2 \text{ ns}
\]

The setup relationship in this example demonstrates that the data is not launched at edge one, and the data that is launched at edge three must be captured; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the launch edge by two clock periods with a start multicycle setup exception of three.

The multicycle exception used to correct the default analysis in this example.
Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
    -setup -start 3
```

The timing diagram for the preferred setup relationship for this example.

**Figure 3-63: Preferred Setup Check Analysis**

The timing diagram for default hold check analysis the TimeQuest analyzer performs for a start multicycle setup value of three.

**Figure 3-64: Default Hold Check Analysis**

The calculation that the TimeQuest analyzer performs to determine the hold check.
Figure 3-65: Hold Check

<table>
<thead>
<tr>
<th>Hold Check 1</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current launch edge – previous latch edge</td>
<td>= 0 ns – 2 ns</td>
<td>= -2 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hold Check 2</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next launch edge – current latch edge</td>
<td>= 5 ns – 12 ns</td>
<td>= -7 ns</td>
</tr>
</tbody>
</table>

In this example, hold check two is too restrictive. The data is launched next by the edge at 10 ns and should check against the data captured by the current latch edge at 12 ns, which does not occur in hold check two. To correct the default analysis, you must use a start multicycle hold exception of one.

A Sample Design with SDC File

An example circuit that includes two clocks, a phase-locked loop (PLL), and other common synchronous design elements helps demonstrate how to constrain your design with an SDC file.

Figure 3-66: TimeQuest Constraint Example

The following SDC file contains basic constraints for the example circuit.

Example 3-5: Basic SDC Constraints

```
# Create clock constraints
create_clock -name clockone -period 10.000 [get_ports {clk1}]
create_clock -name clocktwo -period 10.000 [get_ports {clk2}]
# Create virtual clocks for input and output delay constraints
create_clock -name clockone_ext -period 10.000
create_clock -name clocktwo_ext -period 10.000
derive_pll_clocks
```

The Quartus Prime TimeQuest Timing Analyzer

Send Feedback
# derive clock uncertainty
derive_clock_uncertainty
# Specify that clockone and clocktwo are unrelated by assigning
# them to separate asynchronous groups
set_clock_groups \
-asyncronous \n -group {clockone} \n -group {clocktwo altpll0|altpll_component|auto_generated|pll1|clk[0]}

# set input and output delays
set_input_delay -clock { clockone_ext } -max 4 [get_ports {data1}]
set_input_delay -clock { clockone_ext } -min -1 [get_ports {data1}]
set_input_delay -clock { clockone_ext } -max 4 [get_ports {data2}]
set_input_delay -clock { clockone_ext } -min -1 [get_ports {data2}]
set_output_delay -clock { clocktwo_ext } -max 6 [get_ports {dataout}]
set_output_delay -clock { clocktwo_ext } -min -3 [get_ports {dataout}]

The SDC file contains the following basic constraints you should include for most designs:

- Definitions of `clockone` and `clocktwo` as base clocks, and assignment of those settings to nodes in the design.
- Definitions of `clockone_ext` and `clocktwo_ext` as virtual clocks, which represent clocks driving external devices interfacing with the FPGA.
- Automated derivation of generated clocks on PLL outputs.
- Derivation of clock uncertainty.
- Specification of two clock groups, the first containing `clockone` and its related clocks, the second containing `clocktwo` and its related clocks, and the third group containing the output of the PLL. This specification overrides the default analysis of all clocks in the design as related to each other.
- Specification of input and output delays for the design.

Related Information

- Asynchronous Clock Groups on page 3-21
  For more information about asynchronous clock groups.

**Running the TimeQuest Analyzer**

When you compile a design, the TimeQuest timing analyzer automatically performs multi-corner signoff timing analysis after the Fitter has finished.

- To open the TimeQuest analyzer GUI directly from the Quartus Prime software GUI, click **TimeQuest Timing Analyzer** on the Tools menu.
- To perform or repeat multi-corner timing analysis from the Quartus Prime GUI, click **Processing > Start > Start TimeQuest Timing Analyzer**.
- To perform multi-corner timing analysis from a system command prompt, type `quartus_sta <options><project_name>`.
- To run the TimeQuest analyzer as a stand-alone GUI application, type the following command at the command prompt: `quartus_staw`.
- To run the TimeQuest analyzer in interactive command-shell mode, type the following command at a system command prompt: `quartus_sta -s <options><project_name>`.

The following table lists the command-line options available for the `quartus_sta` executable.
### Table 3-9: Summary of Command-Line Options

<table>
<thead>
<tr>
<th>Command-Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>--help</td>
</tr>
<tr>
<td>-t &lt;script file&gt;</td>
<td>--script=&lt;script file&gt;</td>
</tr>
<tr>
<td>-s</td>
<td>--shell</td>
</tr>
<tr>
<td>--tcl_eval &lt;tcl command&gt;</td>
<td>Evaluates the Tcl command &lt;tcl command&gt;.</td>
</tr>
<tr>
<td>--do_report_timing</td>
<td>For all clocks in the design, run the following commands:</td>
</tr>
<tr>
<td></td>
<td>report_timing -npaths 1 -to_clock $clock</td>
</tr>
<tr>
<td></td>
<td>report_timing -setup -npaths 1 -to_clock $clock</td>
</tr>
<tr>
<td></td>
<td>report_timing -hold -npaths 1 -to_clock $clock</td>
</tr>
<tr>
<td></td>
<td>report_timing -recovery -npaths 1 -to_clock $clock</td>
</tr>
<tr>
<td>--force_dat</td>
<td>Forces an update of the project database with new delay information.</td>
</tr>
<tr>
<td>--lower_priority</td>
<td>Lowers the computing priority of the quartus_sta process.</td>
</tr>
<tr>
<td>--post_map</td>
<td>Uses the post-map database results.</td>
</tr>
<tr>
<td>--sdc=&lt;SDC_file&gt;</td>
<td>Specifies the SDC file to use.</td>
</tr>
<tr>
<td>--report_script=&lt;script&gt;</td>
<td>Specifies a custom report script to call.</td>
</tr>
<tr>
<td>--speed=&lt;value&gt;</td>
<td>Specifies the device speed grade used for timing analysis.</td>
</tr>
<tr>
<td>--tq2pt</td>
<td>Generates temporary files to convert the TimeQuest analyzer SDC file(s) to a PrimeTime SDC file.</td>
</tr>
<tr>
<td>-f &lt;argument file&gt;</td>
<td>Specifies a file containing additional command-line arguments.</td>
</tr>
<tr>
<td>-c &lt;revision name&gt;</td>
<td>--rev=&lt;revision_name&gt;</td>
</tr>
<tr>
<td>--multicorner</td>
<td>Specifies that all slack summary reports be generated for both slow- and fast-corners.</td>
</tr>
<tr>
<td>--multicorner[=on</td>
<td>off]</td>
</tr>
<tr>
<td>--voltage=&lt;value_in_mV&gt;</td>
<td>Specifies the device voltage, in mV used for timing analysis.</td>
</tr>
<tr>
<td>--temperature=&lt;value_in_C&gt;</td>
<td>Specifies the device temperature in degrees Celsius, used for timing analysis.</td>
</tr>
<tr>
<td>--parallel [=&lt;num_processors&gt;]</td>
<td>Specifies the number of computer processors to use on a multiprocessor system.</td>
</tr>
</tbody>
</table>
### Command-Line Option

<table>
<thead>
<tr>
<th>Command-Line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--64bit</td>
<td>Enables 64-bit version of the executable.</td>
</tr>
<tr>
<td>--mode=implement</td>
<td>finalize</td>
</tr>
</tbody>
</table>

**Related Information**

- **Constraining and Analyzing with Tcl Commands** on page 3-65
  For more information about using Tcl commands to constrain and analyze your design
- **Recommended Flow for First Time Users** on page 3-2
  For more information about steps to perform before opening the TimeQuest analyzer.

### Quartus Prime Settings

Within the Quartus Prime software, there are a number of quick steps for setting up your design with TimeQuest. You can modify the appropriate settings in **Assignments > Settings**.

In the **Settings** dialog box, select **TimeQuest Timing Analyzer** in the **Category** list.

The **TimeQuest Timing Analyzer** settings page is where you specify the title and location for a Synopsis Design Constraint (SDC) file. The SDC file is an industry standard format for specifying timing constraints. If no SDC file exists, you can create one based on the instructions in this document. The Quartus Prime software provides an SDC template you can use to create your own.

The following TimeQuest options should be on by default:

- Enable multicorner timing analysis—Directs the TimeQuest analyzer to analyze all the timing models of your FPGA against your constraints. This is required for final timing sign-off. Unchecked, only the slow timing model is be analyzed.
- Enable common clock path pessimism removal—Prevents timing analysis from over-calculating the effects of **On-Die Variation**. This makes timing better, and there really is no reason for this to be disabled.
- Report worst-case paths during compilation—This optional setting displays summary of the worst paths in your timing report. This type of path analysis is covered in more detail later in this document. While useful, this summary can increase the size of the `<project>.sta.rpt` with all of these paths.
- Tcl script file for custom reports—This optional setting should prove useful later, allowing you to add custom reports to create a custom analysis. For example, if you are only working on a portion of the full FPGA, you may want additional timing reports that cover that hierarchy.

**Note:** In addition, certain values are set by default. The default duty-cycle is 50% and the default clock frequency is 1Ghz.

### SDC File Precedence

The Fitter and the TimeQuest analyzer process SDC files in the order you specify in the Quartus Prime Settings File (.qsf). You can add and remove SDC files to process and specify the order they are processed from the **Assignments** menu.

Click **Settings**, then **TimeQuest Timing Analyzer** and add or remove SDC files, or specify a processing order in the **SDC files to include in the project** box. When you create a new SDC file for a project, you must add it to the project for it to be read during fitting and timing analysis. If you use the Quartus Prime Text Editor to create an SDC file, the option to add it to the project is enabled by default when you save the file. If you use any other editor to create an SDC file, you must remember to add it to the project. If no
SDC files are listed in the .qsf, the Quartus Prime software looks for an SDC named <current revision>.sdc in the project directory. When you use IP from Altera, and some third-parties, the SDC files are often included in a project through an intermediate file called a Quartus Prime IP File (.qip). A .qip file points to all source files and constraints for a particular IP. If SDC files for IP blocks in your design are included through with a .qip, do not re-add them manually. An SDC file can also be added from a Quartus Prime IP File (.qip) included in the .qsf.

Figure 3-67: .sdc File Order of Precedence

Is one or more .sdc file specified in the .qsf?
Yes

No

Does an .sdc named <current revision>.sdc exist in the project directory?
Yes

No

Analyze the design

Note: If you type the read_sdc command at the command line without any arguments, the TimeQuest analyzer reads constraints embedded in HDL files, then follows the SDC file precedence order.

The SDC file must contain only SDC commands that specify timing constraints. There are some techniques to control which constraints are applied during different parts of the compilation flow. Tcl commands to manipulate the timing netlist or control the compilation must be in a separate Tcl script.

Understanding Results

Knowing how your constraints are displayed when analyzing a path is one of the most important skills of timing analysis. This information completes your understanding of timing analysis and lets you correlate the SDC input to the back-end analysis, and determine how the delays in the FPGA affect timing.

Iterative Constraint Modification

Sometimes it is useful to change an SDC constraint and reanalyze the timing results. This flow is particularly common when you are creating timing constraints and want to ensure that they will be applied appropriately during compilation and timing analysis.

Use the following steps when you iteratively modify constraints:

1. Open the TimeQuest Timing Analyzer
2. Generate the appropriate reports.
3. Analyze your results
4. Edit your SDC file and save
5. Double-click Reset Design
6. Generate the appropriate reports.
7. Analyze your results
8. Repeat steps 4-7 as necessary.

Open the TimeQuest Timing Analyzer—It is most common to use this interactive approach in the TimeQuest GUI. You can also use the command-line shell mode, but it does not include some of the time-saving automatic features in the GUI.

Generate the appropriate reports — Use the Report All Summaries task under Macros to generate setup, hold, recovery, and removal summaries, as well as minimum pulse width checks, and a list of all the defined clocks. These summaries cover all constrained paths in your design. Especially when you are modifying or correcting constraints, you should also perform the Diagnostic task to create reports to identify unconstrained parts of your design, or ignored constraints. Double-click on any of the report tasks to automatically run the three tasks under Netlist Setup if they haven't already run. One of those tasks reads all SDC files.

Analyze your results—When you are modifying or correcting constraints, review the reports to find any unexpected results. For example, a cross-domain path might indicate that you forgot to cut a transfer by including a clock in a clock group.

Edit your SDC file and save it—Create or edit the appropriate constraints in your SDC files. If you edit your SDC file in the Quartus Prime Tex Editor, you can benefit from tooltips showing constraint options, and dialog boxes that guide you when creating constraints.

Reset the design—Double click Reset Design task to remove all constraints from your design. Removing all constraints from your design prepares it to reread the SDC files, including your changes.

Be aware that this method just performs timing analysis using new constraints, but the fit being analyzed has not changed. The place-and-route was performed with the old constraints, but you are analyzing with new constraints, so if something is failing timing against these new constraints,you may need to run place-and-route again.

For example, the Fitter may concentrate on a very long path in your design, trying to close timing. For example, you may realize that a path runs at a lower rate, and so have added set_multicycle_path assignments to relax the relationship (open the window when data is valid). When you perform TimeQuest analysis iteratively with these new multicycles, new paths replace the old. The new paths may have sub-optimal placement since the Fitter was concentrating on the previous paths when it ran, because they were more critical. The iterative method is recommend for getting your SDC files correct, but you should perform a full compilation to see what the Quartus Prime software can do with those constraints.

Related Information
Relaxing Setup with set_multicycle_path on page 3-31

Set Operating Conditions Dialog Box
You can select different operating conditions to analyze from those used to create the existing timing netlist.

Operating conditions consist of voltage and temperature options that are used together. You can run timing analysis for different delay models without having to delete the existing timing netlist. The TimeQuest analyzer supports multi-corner timing analysis which you can turn on in the dialog box of the command you are performing. A control has been added to the TimeQuest UI where you can select operating conditions and analyze timing for combinations of corners.

Select a voltage/temperature combination and double-click Report Timing under Custom Reports in the Tasks pane.
Reports that fail timing appear in red type, reports that pass appear in black type. Reports that have not yet been run are in gold with a question mark (?). Selecting another voltage/temperature combination creates a new report, but any reports previously run persist.

You can use the following context menu options to generate or regenerate reports in the Report window:

- **Regenerate**—Regenerate the selected report.
- **Generate in All Corners**—Generate a timing report using all corners.
- **Regenerate All Out of Date**—Regenerate all reports.
- **Delete All Out of Date**—Flush all the reports that have been run to clear the way for new reports with modifications to timing.

Each operating condition generates its own set of reports which appear in their own folders under the Reports list. Reports that have not yet been generated display a '?' icon in gold. As each report is generated, the folder is updated with the appropriate output.

**Note:** Reports for a corner not being generated persist until that particular operating condition is modified and a new report is created.
Once you are comfortable with the Report All Summaries command, the next tool in the TimeQuest analyzer toolbox is Report Timing.

The TimeQuest analyzer displays reports in the Report pane, and is similar to a table of contents for all the reports created. Selecting any name in the Report panel displays that report in the main viewing pane. Below is a design with the Summary (Setup) report highlighted:

The main viewing pane shows the Slack for every clock domain. Positive slack is good, saying these paths meet timing by that much. The End Point TNS stands for Total Negative Slack, and is the sum of all slacks for each destination and can be used as a relative assessment of how much a domain is failing.

However, this is just a summary. To get details on any domain, you can right-click that row and select Report Timing.

The Report Timing dialog box appears, auto-filled with the Setup radio button selected and the To Clock box filled with the selected clock. This occurs because you were viewing the Setup Summary report, and right-clicked on that particular clock. As such, the worst 10 paths where that is the destination clock were reported. You can modify the settings in various ways, such as increasing the number of paths to report, adding a Target filter, adding a From Clock, writing the report to a text file, etc.

Note that any report_timing command can be copied from the Console at the bottom into a user-created Tcl file, so that you can analyze specific paths again in the future without having to negotiate the TimeQuest analyzer UI. This is often done as users become more comfortable with TimeQuest and find themselves analyzing the same problematic parts of their design over and over, but is not required. Many complex designs successfully use TimeQuest as a diving tool, i.e. just starting with summaries and diving down into the failing paths after each compile.

Analyzing Results with Report Timing

Report Timing is one of the most useful analysis tools in TimeQuest. Many designs require nothing but this command. In the TimeQuest analyzer, this command can be accessed from the Tasks menu, from the Reports > Custom Reports menu, or by right-clicking on nodes or assignments in TimeQuest.

You can review all of the options for Report Timing by typing report_timing -long_help in the TimeQuest console.

Clocks

The From Clock and To Clock in the Clocks box are used to filter paths where the selected clock is used as the launch or latch. The pull-down menu allows you to choose from existing clocks (although admittedly has a ”limited view” for long clock names).

Targets

The boxes in the Targets box are targets for the From Clock and To Clock settings, and allow you to report paths with only particular endpoints. These are usually filled with register names or I/O ports, and can be wildcarded. For example, you might use the following to only report paths within a hierarchy of interest:

```
report_timing -from *|egress:egress_inst|* -to *|egress:egress_inst|* -(other options)
```

If the From, To, or Through boxes are empty, then the TimeQuest analyzer assumes you are referring to all possible targets in the device, which can also be represented with a wildcard (*). The From and To options cover the majority of situations. The Through option is used to limit the report for paths that pass through combinatorial logic, or a particular pin on a cell. This is seldom used, and may not be very reliable due to combinatorial node name changes during synthesis. Clicking the browse Browse box after...
each target opens the Name Finder dialog box to search for specific names. This is especially useful to make sure the name being entered matches nodes in the design, since the Name Finder can immediately show what matches a user’s wildcard.

**Analysis type**

The Analysis type options are Setup, Hold, Recovery, or Removal. These will be explained in more detail later, as understanding them is the underpinning of timing analysis.

**Output**

The Detail level, is an option often glanced over that should be understood. It has four options, but I will only discuss three.

The first level is called Summary, and produces a report which only displays Summary information such as

- Slack
- From Node
- To Node
- Launch Clock
- Latch Clock
- Relationship
- Clock Skew
- Data Delay

The Summary report is always reported with more detailed reports, so the user would choose this if they want less info. A good use for summary detail is when writing the report to a text file, where Summary can be quite brief.

The next level is Path only. This report displays all the detailed information, except the Data Path tab displays the clock tree as one line item. This is useful when you know the clock tree is correct, details are not relevant. This is common for most paths within the FPGA. A useful data point is to look at the Clock Skew column in the Summary report, and if it’s a small number, say less than +/-150ps, then the clock tree is well balanced between source and destination.

If there is clock skew, you should select the Full path option. This breaks the clock tree out into explicit detail, showing every cell it goes through, including such things as the input buffer, PLL, global buffer (called CLKCTRL_), and any logic. If there is clock skew, this is where you can determine what is causing the clock skew in your design. The Full path option is also recommended for I/O analysis, since only the source clock or destination clock is inside the FPGA, and therefore its delay plays a critical role in meeting timing.

The Data Path tab of a detailed report gives the delay break-downs, but there is also useful information in the Path Summary and Statistics tabs, while the Waveform tab is useful to help visualize the Data Path analysis. I would suggest taking a few minutes to look at these in the user’s design. The whole analysis takes some time to get comfortable with, but hopefully is clear in what it’s doing.

Enable multi corner reports allows you to enable or disable multi-corner timing analysis. This option is on by default.

Report Timing also has the Report panel name, which displays the name used in TimeQuest’s Report section. There is also an optional File name switch, which allows you to write the information to a file. If you append .htm as a suffix, the TimeQuest analyzer produces the report as HTML. The File options radio buttons allow you to choose between Overwrite and Append when saving the file.
Paths

The default value for **Report number of paths** is 10. Two endpoints may have a lot of combinatorial logic between them and might have many different paths. Likewise, a single destination may have hundreds of paths leading to it. Because of this, you might list hundreds of paths, many of which have the same destination and might have the same source. By turning on **Pairs only** you can list only one path for each pair of source and destination. An even more powerful way to filter the report is limit the Maximum number of paths per endpoints. You can also filter paths by entering a value in the **Maximum slack limit** field.

Tcl command

Finally, at the bottom is the **Tcl command** field, which displays the Tcl syntax of what is run in TimeQuest. You can edit this directly before running the **Report Timing** command.

**Note:**

A useful addition is to add the `-false_path` option to the command line string. With this option, only false paths are listed. A false path is any path where the launch and latch clock have been defined, but the path was cut with either a `set_false_path` assignment or `set_clock_groups_assignment`. Paths where the launch or latch clock was never constrained are not considered false paths. This option is useful to see if a false path assignment worked and what paths it covers, or to look for paths between clock domains that should not exist. The **Task** window’s **Report False Path** custom report is nothing more than **Report Timing** with the `-false_path` flag enabled.

**Correlating Constraints to the Timing Report**

A critical part of timing analysis is how timing constraints appear in the **Report Timing** analysis. Most constraints only affect the launch and latch edges. Specifically, `create_clock` and `create_generated_clock` create clocks with default relationships. The command `set_multicycle_path` will modify those default relationships, while `set_max_delay` and `set_min_delay` are low-level overrides that explicitly tell TimeQuest what the launch and latch edges should be.

The following figures are from an example of the output of **Report Timing** on a particular path.

Initially, the design features a clock driving the source and destination registers with a period of 10ns. This results in a setup relationship of 10ns (launch edge = 0ns, latch edge = 10ns) and hold relationship of 0ns (launch edge = 0ns, latch edge = 0ns) from the command:

```
create_clock -name clocktwo -period 10.000 [get_ports {clk2}]
```
In the next figure, using `set_multicycle_path` adds multicycles to relax the setup relationship, or open the window, making the setup relationship 20ns while the hold relationship is still 0ns:

```
set_multicycle_path -from clocktwo -to clocktwo -setup -end 2
set_multicycle_path -from clocktwo -to clocktwo -hold -end 1
```
In the last figure, using the `set_max_delay` and `set_min_delay` constraints lets you explicitly override the relationships. Note that the only thing changing for these different constraints is the Launch Edge Time and Latch Edge Times for setup and hold analysis. Every other line item comes from delays inside the FPGA and are static for a given fit. Whenever analyzing how your constraints affect the timing requirements, this is the place to look.
For I/O, this all holds true except we must add in the \(-\text{max}\) and \(-\text{min}\) values. They are displayed as \(i\text{Ext}\) or \(o\text{Ext}\) in the \textbf{Type} column. An example would be an output port with a `set_output_delay -max 1.0` and `set_output_delay -min -0.5`:

Once again, the launch and latch edge times are determined by the clock relationships, multicycles and possibly `set_max_delay` or `set_min_delay` constraints. The value of `set_output_delay` is also added in as an \(o\text{Ext}\) value. For outputs this value is part of the \textbf{Data Required Path}, since this is the external part of the analysis. The setup report on the left will subtract the \(-\text{max}\) value, making the setup relationship harder to meet, since we want the \textbf{Data Arrival Path} to be shorter than the \textbf{Data Required Path}. The \(-\text{min}\) value is also subtracted, which is why a negative number makes hold timing more restrictive, since we want the \textbf{Data Arrival Path} to be longer than the \textbf{Data Required Path}.

Related Information

- Relaxing Setup with `set_multicycle_path` on page 3-31

### Constraining and Analyzing with Tcl Commands

You can use Tcl commands from the Quartus Prime software Tcl Application Programming Interface (API) to constrain, analyze, and collect information for your design. This section focuses on executing timing analysis tasks with Tcl commands; however, you can perform many of the same functions in the TimeQuest analyzer GUI. SDC commands are Tcl commands for constraining a design. SDC extension commands provide additional constraint methods and are specific to the TimeQuest analyzer. Additional TimeQuest analyzer commands are available for controlling timing analysis and reporting. These commands are contained in the following Tcl packages available in the Quartus Prime software:

- ::quartus::sta
- ::quartus::sdc
- ::quartus::sdc_ext
**Related Information**

- **::quartus::sta**
  For more information about TimeQuest analyzer Tcl commands and a complete list of commands, refer to Quartus Prime Help.

- **::quartus::sdc**
  For more information about standard SDC commands and a complete list of commands, refer to Quartus Prime Help.

- **::quartus::sdc_ext**
  For more information about Altera extensions of SDC commands and a complete list of commands, refer to Quartus Prime Help.

**Collection Commands**

The TimeQuest analyzer Tcl commands often return data in an object called a collection. In your Tcl scripts you can iterate over the values in collections to access data contained in them. The software returns collections instead of Tcl lists because collections are more efficient than lists for large sets of data.

The TimeQuest analyzer supports collection commands that provide easy access to ports, pins, cells, or nodes in the design. Use collection commands with any constraints or Tcl commands specified in the TimeQuest analyzer.

**Table 3-10: SDC Collection Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description of the collection returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_clocks</td>
<td>All clocks in the design.</td>
</tr>
<tr>
<td>all_inputs</td>
<td>All input ports in the design.</td>
</tr>
<tr>
<td>all_outputs</td>
<td>All output ports in the design.</td>
</tr>
<tr>
<td>all_registers</td>
<td>All registers in the design. All cell names in the collection match the specified pattern. Wildcards can be used to select multiple cells at the same time.</td>
</tr>
<tr>
<td>get_cells</td>
<td>Cells in the design. All net names in the collection match the specified pattern. You can use wildcards to select multiple nets at the same time.</td>
</tr>
<tr>
<td>get_clocks</td>
<td>Lists clocks in the design. When used as an argument to another command, such as the -from or -to of set_multicycle_path, each node in the clock represents all nodes clocked by the clocks in the collection. The default uses the specific node (even if it is a clock) as the target of a command.</td>
</tr>
<tr>
<td>get_nets</td>
<td>Nets in the design. All net names in the collection match the specified pattern. You can use wildcards to select multiple nets at the same time.</td>
</tr>
<tr>
<td>get_pins</td>
<td>Pins in the design. All pin names in the collection match the specified pattern. You can use wildcards to select multiple pins at the same time.</td>
</tr>
<tr>
<td>get_ports</td>
<td>Ports (design inputs and outputs) in the design.</td>
</tr>
</tbody>
</table>

You can also examine collections and experiment with collections using wildcards in the TimeQuest analyzer by clicking **Name Finder** from the **View** menu.
**Wildcard Characters**

To apply constraints to many nodes in a design, use the “*” and “?” wildcard characters. The “*” wildcard character matches any string; the “?” wildcard character matches any single character.

If you make an assignment to node `reg*`, the TimeQuest analyzer searches for and applies the assignment to all design nodes that match the prefix `reg` with any number of following characters, such as `reg, reg1, reg[2], regbank, and reg12bank`.

If you make an assignment to a node specified as `reg?`, the TimeQuest analyzer searches and applies the assignment to all design nodes that match the prefix `reg` and any single character following; for example, `reg1, rega, and reg4`.

**Adding and Removing Collection Items**

Wildcards used with collection commands define collection items identified by the command. For example, if a design contains registers named `src0, src1, src2, and dst0`, the collection command `[get_registers src*]` identifies registers `src0, src1, and src2, but not register dst0. To identify register dst0, you must use an additional command, `[get_registers dst*]`. To include dst0, you could also specify a collection command `[get_registers {src* dst*}]`.

To modify collections, use the `add_to_collection` and `remove_from_collection` commands. The `add_to_collection` command allows you to add additional items to an existing collection.

**add_to_collection Command**

```bash
add_to_collection <first collection> <second collection>
```

*Note:* The `add_to_collection` command creates a new collection that is the union of the two specified collections.

The `remove_from_collection` command allows you to remove items from an existing collection.

**remove_from_collection Command**

```bash
remove_from_collection <first collection> <second collection>
```

You can use the following code as an example for using `add_to_collection` for adding items to a collection.

**Adding Items to a Collection**

```bash
#Setting up initial collection of registers
set regs1 [get_registers a*]
#Setting up initial collection of keepers
set kprs1 [get_keepers b*]
#Creating a new set of registers of $regs1 and $kprs1
set regs_union [add_to_collection $kprs1 $regs1]
#OR
#Creating a new set of registers of $regs1 and b*
#Note that the new collection appends only registers with name b*
# not all keepers
set regs_union [add_to_collection $regs1 b*]
```

In the Quartus Prime software, keepers are I/O ports or registers. A SDC file that includes `get_keepers` can only be processed as part of the TimeQuest analyzer flow and is not compatible with third-party timing analysis flows.
Related Information

- **add_to_collection**
- **remove_from_collection**

For more information about the `add_to_collection` and `remove_from_collection` commands, refer to Quartus Prime Help.

**Getting Other Information about Collections**

You can display the contents of a collection with the `query_collection` command. Use the `-report_format` option to return the contents in a format of one element per line. The `-list_format` option returns the contents in a Tcl list.

```
query_collection -report_format -all $regs_union
```

Use the `get_collection_size` command to return the size of a collection; the number of items it contains. If your collection is in a variable named `col`, it is more efficient to use `set num_items [get_collection_size $col]` than `set num_items [llength [query_collection -list_format $col]]`

**Using the get_pins Command**

The `get_pins` command supports options that control the matching behavior of the wildcard character (`*`). Depending on the combination of options you use, you can make the wildcard character (`*`) respect or ignore individual levels of hierarchy, which are indicated by the pipe character (`|`). By default, the wildcard character (`*`) matches only a single level of hierarchy.

These examples filter the following node and pin names to illustrate function:

- `foo` (a hierarchy level named `foo`)
- `foo|dataa` (an input pin in the instance `foo`)
- `foo|datab` (an input pin in the instance `foo`)
- `foo|bar` (a combinational node named `bar` in the `foo` instance)
- `foo|bar|datac` (an input pin to the combinational node named `bar`)
- `foo|bar|datad` (an input pin to the combinational node `bar`)

**Table 3-11: Sample Search Strings and Search Results**

<table>
<thead>
<tr>
<th>Search String</th>
<th>Search Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`get_pins *</td>
<td>dataa`</td>
</tr>
<tr>
<td>`get_pins *</td>
<td>datac`</td>
</tr>
<tr>
<td>`get_pins *</td>
<td>*</td>
</tr>
<tr>
<td>`get_pins foo</td>
<td>*`</td>
</tr>
<tr>
<td>`get_pins -hierarchical *</td>
<td>*</td>
</tr>
<tr>
<td>`get_pins -hierarchical foo</td>
<td>*`</td>
</tr>
<tr>
<td>`get_pins -hierarchical *</td>
<td>datac`</td>
</tr>
</tbody>
</table>

(3) The search result is `<empty>` because the wildcard character (`*`) does not match more than one hierarchy level, indicated by a pipe character (`|`), by default. This command would match any pin named `datac` in instances at the top level of the design.
The default method separates hierarchy levels of instances from nodes and pins with the pipe character (|). A match occurs when the levels of hierarchy match, and the string values including wildcards match the instance and/or pin names. For example, the command `get_pins <instance_name>|*|datac` returns all the `datac` pins for registers in a given instance. However, the command `get_pins *|datac` returns an empty collection because the levels of hierarchy do not match.

Use the `-hierarchical` matching scheme to return a collection of cells or pins in all hierarchies of your design.

For example, the command `get_pins -hierarchical *|datac` returns all the `datac` pins for all registers in your design. However, the command `get_pins -hierarchical *|*|datac` returns an empty collection because more than one pipe character (|) is not supported.

The `-compatibility_mode` option returns collections matching wildcard strings through any number of hierarchy levels. For example, an asterisk can match a pipe character when using `-compatibility_mode`.

### Identifying the Quartus Prime Software Executable from the SDC File

To identify which Quartus Prime software executable is currently running you can use the `$::TimeQuestInfo(nameofexecutable)` variable from within an SDC file. This technique is most commonly used when you want to use an overconstraint to cause the Fitter to work harder on a particular path or set of paths in the design.

#### Identifying the Quartus Prime Executable

```bash
Identify which executable is running:
set current_exe $::TimeQuestInfo(nameofexecutable)
if { [string equal $current_exe "quartus_fit"] } {
    #Apply .sdc assignments for Fitter executable here
} else {
    #Apply .sdc assignments for non-Fitter executables here
}
if { ![string equal "quartus_sta" $::TimeQuestInfo(nameofexecutable)] } {
    #Apply .sdc assignments for non-TimeQuest executables here
} else {
    #Apply .sdc assignments for TimeQuest executable here
}
```

Examples of different executable names are `quartus_map` for Analysis & Synthesis, `quartus_fit` for Fitter, and `quartus_sta` for the TimeQuest analyzer.

### Locating Timing Paths in Other Tools

You can locate paths and elements from the TimeQuest analyzer to other tools in the Quartus Prime software.

---

*When you use `-compatibility_mode`, pipe characters (|) are not treated as special characters when used with wildcards.*
Use the **Locate** or **Locate Path** command in the TimeQuest analyzer GUI or the *locate* command in the Tcl console in the TimeQuest analyzer GUI. Right-click on most paths or node names in the TimeQuest analyzer GUI to access the **Locate** or **Locate Path** options.

The following commands are examples of how to locate the ten paths with the worst timing slack from TimeQuest analyzer to the **Technology Map Viewer** and locate all ports matching data* in the **Chip Planner**.

**Example 3-6: Locating from the TimeQuest Analyzer**

```
# Locate in the Technology Map Viewer the ten paths with the worst slack
locate [get_timing_paths -npaths 10] -tmv
# locate all ports that begin with data in the Chip Planner
locate [get_ports data*] -chip
```

Related Information

- **Viewing Timing Analysis Results**
  - For more information about locating paths from the TimeQuest analyzer, refer to Quartus Prime Help.
- **locate**
  - For more information on this command, refer to Quartus Prime Help.

### Generating Timing Reports

The TimeQuest analyzer provides real-time static timing analysis result reports. The TimeQuest analyzer does not automatically generate most reports; you must create each report individually in the TimeQuest analyzer GUI or with command-line commands. You can customize in which report to display specific timing information, excluding fields that are not required.

Some of the different command-line commands you can use to generate reports in the TimeQuest analyzer and the equivalent reports shown in the TimeQuest analyzer GUI.

**Table 3-12: TimeQuest Analyzer Reports**

<table>
<thead>
<tr>
<th>Command-Line Command</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>report_timing</td>
<td>Timing report</td>
</tr>
<tr>
<td>report_exceptions</td>
<td>Exceptions report</td>
</tr>
<tr>
<td>report_clock_transfers</td>
<td>Clock Transfers report</td>
</tr>
<tr>
<td>report_min_pulse_width</td>
<td>Minimum Pulse Width report</td>
</tr>
<tr>
<td>report_ucp</td>
<td>Unconstrained Paths report</td>
</tr>
</tbody>
</table>

During compilation, the Quartus Prime software generates timing reports on different timing areas in the design. You can configure various options for the TimeQuest analyzer reports generated during compilation.

You can also use the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` assignment to generate a report of the worst-case timing paths for each clock domain. This report contains worst-case timing data for setup, hold, recovery, removal, and minimum pulse width checks.
Use the `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignment to specify the number of paths to report for each clock domain.

An example of how to use the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` and `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignments in the `.qsf` to generate reports.

### Generating Worst-Case Timing Reports

```plaintext
#Enable Worst-Case Timing Report
set_global_assignment -name TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS ON
#Report 10 paths per clock domain
set_global_assignment -name TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS 10
```

### Fmax Summary Report panel

Fmax Summary Report panel lists the maximum frequency of each clock in your design. In some designs you may see a note indicating "Limit due to hold check. Typically, Fmax is not limited by hold checks, because they are often same-edge relationships, and therefore independent of clock frequency, for example, launch = 0, latch = 0. However, if you have an inverted clock transfer, or a multicycle transfer such as setup=2, hold=0, then the hold relationship is no longer a same-edge transfer and changes as the clock frequency changes. The value in the Restricted Fmax column incorporates limits due to hold time checks in the situations described previously, as well as minimum period and pulse width checks. If hold checks limit the Fmax more than setup checks, that is indicated in the Note: column as "Limit due to hold check".

**Related Information**

- [::quartus::sta](#)
  For more information on this command, refer to Quartus Prime Help.
- [TimeQuest Timing Analyzer Page](#)
  For more information about the options you can set to customize TimeQuest analyzer reports.
- [Area and Timing Optimization](#)
  For more information about timing closure recommendations.

## Document Revision History

### Table 3-13: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>- Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added a description of running three- and four-corner analysis with --mode=implement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Added description for new <code>set_operating_conditions</code> UI.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| 2015.05.04  | 15.0.0  | Added and updated contents in support of new timing algorithms for Arria 10:  
  - Enhanced Timing Analysis for Arria 10  
  - Maximum Skew (set_max_skew command)  
  - Net Delay (set_net_delay command)  
  - Create Generated Clocks (clock-as-data example) |
| 2014.12.15  | 14.1.0  | Major reorganization. Revised and added content to the following topic areas:  
  - Timing Constraints  
  - Create Clocks and Clock Constraints  
  - Creating Generated Clocks  
  - Creating Clock Groups  
  - Clock Uncertainty  
  - Running the TimeQuest Analyzer  
  - Generating Timing Reports  
  - Understanding Results  
  - Constraining and Analyzing with Tcl Commands |
| August 2014 | 14.0a10.0 | Added command line compilation requirements for Arria 10 devices. |
| June 2014   | 14.0.0  |  
  - Minor updates.  
  - Updated format. |
| November 2013 | 13.1.0 |  
  - Removed HardCopy device information. |
| June 2012   | 12.0.0  |  
  - Reorganized chapter.  
  - Added “Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor” section on creating an SDC constraints file with the Insert Template dialog box.  
  - Added “Identifying the Quartus Prime Software Executable from the SDC File” section.  
  - Revised multicycle exceptions section. |
| November 2011 | 11.1.0 |  
  - Consolidated content from the Best Practices for the Quartus Prime TimeQuest Timing Analyzer chapter.  
  - Changed to new document template. |
| May 2011    | 11.0.0  |  
  - Updated to improve flow. Minor editorial updates. |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| December 2010| 10.1.0  | • Changed to new document template.  
|              |         | • Revised and reorganized entire chapter.  
|              |         | • Linked to Quartus Prime Help. |
| July 2010    | 10.0.0  | Updated to link to content on SDC commands and the TimeQuest analyzer GUI in Quartus Prime Help. |
| November 2009| 9.1.0   | Updated for the Quartus Prime software version 9.1, including:  
|              |         | • Added information about commands for adding and removing items from collections  
|              |         | • Added information about the set_timing_derate and report_skew commands  
|              |         | • Added information about worst-case timing reporting  
|              |         | • Minor editorial updates |
| November 2008| 8.1.0   | Updated for the Quartus Prime software version 8.1, including:  
|              |         | • Added the following sections:  
|              |         | “set_net_delay” on page 7–42  
|              |         | “Annotated Delay” on page 7–49  
|              |         | “report_net_delay” on page 7–66  
|              |         | • Updated the descriptions of the -append and -file <name> options in tables throughout the chapter  
|              |         | • Updated entire chapter using 8½” × 11” chapter template  
|              |         | • Minor editorial updates |

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
The PowerPlay Power Analysis tools allow you to estimate device power consumption accurately.

As designs grow larger and process technology continues to shrink, power becomes an increasingly important design consideration. When designing a PCB, you must estimate the power consumption of a device accurately to develop an appropriate power budget, and to design the power supplies, voltage regulators, heat sink, and cooling system.

The following figure shows the PowerPlay Power Analysis tools ability to estimate power consumption from early design concept through design implementation.

Figure 4-1: PowerPlay Power Analysis From Design Concept Through Design Implementation

For the majority of the designs, the PowerPlay Power Analyzer and the PowerPlay EPE spreadsheet have the following accuracy after the power models are final:

- PowerPlay Power Analyzer — ±20% from silicon, assuming that the PowerPlay Power Analyzer uses the Value Change Dump File (.vcd) generated toggle rates.
- PowerPlay EPE spreadsheet — ±20% from the PowerPlay Power Analyzer results using .vcd generated toggle rates. 90% of EPE designs (using .vcd generated toggle rates exported from PPPA) are within ±30% silicon.
The toggle rates are derived using the PowerPlay Power Analyzer with a .vcd file generated from a gate level simulation representative of the system operation.

Related Information
- PowerPlay Early Power Estimators (EPE) and Power Analyzer

Types of Power Analyses

Understanding the uses of power analysis and the factors affecting power consumption helps you to use the PowerPlay Power Analyzer effectively. Power analysis meets the following significant planning requirements:

- **Thermal planning**—Thermal power is the power that dissipates as heat from the FPGA. You must use a heatsink or fan to act as a cooling solution for your device. The cooling solution must be sufficient to dissipate the heat that the device generates. The computed junction temperature must fall within normal device specifications.
- **Power supply planning**—Power supply is the power needed to run your device. Power supplies must provide adequate current to support device operation.

  **Note:** For power supply planning, use the PowerPlay EPE at the early stages of your design cycle. Use the PowerPlay Power Analyzer reports when your design is complete to get an estimate of your design power requirement.

The two types of analyses are closely related because much of the power supplied to the device dissipates as heat from the device; however, in some situations, the two types of analyses are not identical. For example, if you use terminated I/O standards, some of the power drawn from the power supply of the device dissipates in termination resistors rather than in the device.

Power analysis also addresses the activity of your design over time as a factor that impacts the power consumption of the device. The static power ($P_{\text{STATIC}}$) is the thermal power dissipated on chip, independent of user clocks. $P_{\text{STATIC}}$ includes the leakage power from all FPGA functional blocks, except for I/O DC bias power and transceiver DC bias power, which are accounted for in the I/O and transceiver sections. Dynamic power is the additional power consumption of the device due to signal activity or toggling.

Related Information
- PowerPlay Early Power Estimator (EPE) User Guide

Differences between the PowerPlay EPE and the *Quartus Prime* PowerPlay Power Analyzer

The following table lists the differences between the PowerPlay EPE and the Quartus Prime PowerPlay Power Analyzer.
Table 4-1: Comparison of the PowerPlay EPE and Quartus Prime PowerPlay Power Analyzer

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>PowerPlay EPE</th>
<th>Quartus Prime PowerPlay Power Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase in the design cycle</td>
<td>Any time, but it is recommended to use Quartus Prime PowerPlay Power Analyzer for post-fit power analysis.</td>
<td>Post-fit</td>
</tr>
<tr>
<td>Tool requirements</td>
<td>Spreadsheet program</td>
<td>The Quartus Prime software</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Medium</td>
<td>Medium to very high</td>
</tr>
<tr>
<td>Data inputs</td>
<td>• Resource usage estimates&lt;br&gt;• Clock requirements&lt;br&gt;• Environmental conditions&lt;br&gt;• Toggle rate</td>
<td>• Post-fit design&lt;br&gt;• Clock requirements&lt;br&gt;• Signal activity defaults&lt;br&gt;• Environmental conditions&lt;br&gt;• Register transfer level (RTL) simulation results (optional)&lt;br&gt;• Post-fit simulation results (optional)&lt;br&gt;• Signal activities per node or entity (optional)</td>
</tr>
<tr>
<td>Data outputs (1)</td>
<td>• Total thermal power dissipation&lt;br&gt;• Thermal static power&lt;br&gt;• Thermal dynamic power&lt;br&gt;• Off-chip power dissipation&lt;br&gt;• Current drawn from voltage supplies</td>
<td>• Total thermal power&lt;br&gt;• Thermal static power&lt;br&gt;• Thermal dynamic power&lt;br&gt;• Thermal I/O power&lt;br&gt;• Thermal power by design hierarchy&lt;br&gt;• Thermal power by block type&lt;br&gt;• Thermal power dissipation by clock domain&lt;br&gt;• Off-chip (non-thermal) power dissipation&lt;br&gt;• Device supply currents</td>
</tr>
</tbody>
</table>

The result of the PowerPlay Power Analyzer is only an estimation of power. Altera does not recommend using the result as a specification. The purpose of the estimation is to help you establish guidelines for the power budget of your design. It is important that you verify the actual power during device operation as the information is sensitive to the actual device design and the environmental operating conditions.

(5) PowerPlay EPE and PowerPlay Power Analyzer outputs vary by device family. For more information, refer to the device-specific PowerPlay Early Power Estimators (EPE) and Power Analyzer Page and PowerPlay Power Analyzer Reports in the Quartus Prime Help.
Factors Affecting Power Consumption

Understanding the following factors that affect power consumption allows you to use the PowerPlay Power Analyzer and interpret its results effectively:

- **Device Selection**
- **Environmental Conditions**
- **Device Resource Usage**
- **Signal Activities**

**Device Selection**

Device families have different power characteristics. Many parameters affect the device family power consumption, including choice of process technology, supply voltage, electrical design, and device architecture.

Power consumption also varies in a single device family. A larger device consumes more static power than a smaller device in the same family because of its larger transistor count. Dynamic power can also increase with device size in devices that employ global routing architectures.

The choice of device package also affects the ability of the device to dissipate heat. This choice can impact your required cooling solution choice to comply to junction temperature constraints.

Process variation can affect power consumption. Process variation primarily impacts static power because sub-threshold leakage current varies exponentially with changes in transistor threshold voltage. Therefore, you must consult device specifications for static power and not rely on empirical observation. Process variation has a weak effect on dynamic power.

**Environmental Conditions**

Operating temperature primarily affects device static power consumption. Higher junction temperatures result in higher static power consumption. The device thermal power and cooling solution that you use must result in the device junction temperature remaining within the maximum operating range for the device. The main environmental parameters affecting junction temperature are the cooling solution and ambient temperature.

The following table lists the environmental conditions that could affect power consumption.
### Table 4-2: Environmental Conditions that Could Affect Power Consumption

<table>
<thead>
<tr>
<th>Environmental Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airflow</td>
<td>A measure of how quickly the device removes heated air from the vicinity of the device and replaces it with air at ambient temperature. You can either specify airflow as “still air” when you are not using a fan, or as the linear feet per minute rating of the fan in the system. Higher airflow decreases thermal resistance.</td>
</tr>
<tr>
<td>Heat Sink and Thermal Compound</td>
<td>A heat sink allows more efficient heat transfer from the device to the surrounding area because of its large surface area exposed to the air. The thermal compound that interfaces the heat sink to the device also influences the rate of heat dissipation. The case-to-ambient thermal resistance ($\theta_{CA}$) parameter describes the cooling capacity of the heat sink and thermal compound employed at a given airflow. Larger heat sinks and more effective thermal compounds reduce $\theta_{CA}$.</td>
</tr>
<tr>
<td>Junction Temperature</td>
<td>The junction temperature of a device is equal to: $T_{\text{Junction}} = T_{\text{Ambient}} + P_{\text{Thermal}} \cdot \theta_{JA}$ in which $\theta_{JA}$ is the total thermal resistance from the device transistors to the environment, having units of degrees Celsius per watt. The value $\theta_{JA}$ is equal to the sum of the junction-to-case (package) thermal resistance ($\theta_{JC}$), and the case-to-ambient thermal resistance ($\theta_{CA}$) of your cooling solution.</td>
</tr>
<tr>
<td>Board Thermal Model</td>
<td>The junction-to-board thermal resistance ($\theta_{JB}$) is the thermal resistance of the path through the board, having units of degrees Celsius per watt. To compute junction temperature, you can use this board thermal model along with the board temperature, the top-of-chip $\theta_{JA}$ and ambient temperatures.</td>
</tr>
</tbody>
</table>

### Device Resource Usage

The number and types of device resources used greatly affects power consumption.
- **Number, Type, and Loading of I/O Pins**—Output pins drive off-chip components, resulting in high-load capacitance that leads to a high-dynamic power per transition. Terminated I/O standards require external resistors that draw constant (static) power from the output pin.

- **Number and Type of Hard Logic Blocks**—A design with more logic elements (LEs), multiplier elements, memory blocks, transceiver blocks or HPS system tends to consume more power than a design with fewer circuit elements. The operating mode of each circuit element also affects its power consumption. For example, a DSP block performing $18 \times 18$ multiplications and a DSP block performing multiply-accumulate operations consume different amounts of dynamic power because of different amounts of charging internal capacitance on each transition. The operating mode of a circuit element also affects static power.

- **Number and Type of Global Signals**—Global signal networks span large portions of the device and have high capacitance, resulting in significant dynamic power consumption. The type of global signal is important as well. For example, Stratix V devices support global clocks and quadrant (regional) clocks. Global clocks cover the entire device, whereas quadrant clocks only span one-fourth of the device. Clock networks that span smaller regions have lower capacitance and tend to consume less power. The location of the logic array blocks (LABs) driven by the clock network can also have an impact because the Quartus Prime software automatically disables unused branches of a clock.

### Signal Activities

The behavior of each signal in your design is an important factor in estimating power consumption. The following table lists the two vital behaviors of a signal, which are toggle rate and static probability:

**Table 4-3: Signal Behavior**

<table>
<thead>
<tr>
<th>Signal Behavior</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Toggle rate** | - The toggle rate of a signal is the average number of times that the signal changes value per unit of time. The units for toggle rate are transitions per second and a transition is a change from 1 to 0, or 0 to 1.  
  - Dynamic power increases linearly with the toggle rate as you charge the board trace model more frequently for logic and routing. The Quartus Prime software models full rail-to-rail switching. For high toggle rates, especially on circuit output I/O pins, the circuit can transition before fully charging the downstream capacitance. The result is a slightly conservative prediction of power by the PowerPlay Power Analyzer. |
| **Static probability** | - The static probability of a signal is the fraction of time that the signal is logic 1 during the period of device operation that is being analyzed. Static probability ranges from 0 (always at ground) to 1 (always at logic-high).  
  - Static probabilities of their input signals can sometimes affect the static power that routing and logic consume. This effect is due to state-dependent leakage and has a larger effect on smaller process geometries. The Quartus Prime software models this effect on devices at 90 nm or smaller if it is important to the power estimate. The static power also varies with the static probability of a logic 1 or 0 on the I/O pin when output I/O standards drive termination resistors. |
Note: To get accurate results from the power analysis, the signal activities for analysis must represent the actual operating behavior of your design. Inaccurate signal toggle rate data is the largest source of power estimation error.

**PowerPlay Power Analyzer Flow**

The PowerPlay Power Analyzer supports accurate power estimations by allowing you to specify the important design factors affecting power consumption. The following figure shows the high-level PowerPlay Power Analyzer flow.

**Figure 4-2: PowerPlay Power Analyzer High-Level Flow**

To obtain accurate I/O power estimates, the PowerPlay Power Analyzer requires you to synthesize your design and then fit your design to the target device. You must specify the electrical standard on each I/O cell and the board trace model on each I/O standard in your design.

**Operating Settings and Conditions**

You can specify device power characteristics, operating voltage conditions, and operating temperature conditions for power analysis in the Quartus Prime software.

On the **Operating Settings and Conditions** page of the **Settings** dialog box, you can specify whether the device has typical power consumption characteristics or maximum power consumption characteristics.

On the **Voltage** page of the **Settings** dialog box, you can view the operating voltage conditions for each power rail in the device, and specify supply voltages for power rails with selectable supply voltages.
Note: The Quartus Prime Fitter may override some of the supply voltages settings specified in this chapter. For example, supply voltages for several Stratix V transceiver power supplies depend on the data rate used. If the Fitter detects that voltage required is different from the one specified in the Voltage page, it will automatically set the correct voltage for relevant rails. The Quartus Prime PowerPlay Power Analyzer uses voltages selected by the Fitter if they conflict with the settings specified in the Voltage page.

On the Temperature page of the Settings dialog box, you can specify the thermal operating conditions of the device.

Related Information

- Operating Settings and Conditions Page (Settings Dialog Box)
- Voltage Page (Settings Dialog Box)
- Temperature Page (Settings Dialog Box)

Signal Activities Data Sources

The PowerPlay Power Analyzer provides a flexible framework for specifying signal activities. The framework reflects the importance of using representative signal-activity data during power analysis. Use the following sources to provide information about signal activity:

- Simulation results
- User-entered node, entity, and clock assignments
- User-entered default toggle rate assignment
- Vectorless estimation

The PowerPlay Power Analyzer allows you to mix and match the signal-activity data sources on a signal-by-signal basis. The following figure shows the priority scheme applied to each signal.

Figure 4-3: Signal-Activity Data Source Priority Scheme
Simulation Results

The PowerPlay Power Analyzer directly reads the waveforms generated by a design simulation. Static probability and toggle rate can be calculated for each signal from the simulation waveform. Power analysis is most accurate when you use representative input stimuli to generate simulations.

The PowerPlay Power Analyzer reads results generated by the following simulators:

- ModelSim®
- ModelSim-Altera
- QuestaSim
- Active-HDL
- NCSim
- VCS
- VCS MX
- Riviera-PRO

Signal activity and static probability information are derived from a Verilog Value Change Dump File (.vcd). For more information, refer to Signal Activities on page 4-6.

For third-party simulators, use the EDA Tool Settings to specify the Generate Value Change Dump (VCD) file script option in the Simulation page of the Settings dialog box. These scripts instruct the third-party simulators to generate a .vcd that encodes the simulated waveforms. The Quartus Prime PowerPlay Power Analyzer reads this file directly to derive the toggle rate and static probability data for each signal.

Third-party EDA simulators, other than those listed, can generate a .vcd that you can use with the PowerPlay Power Analyzer. For those simulators, you must manually create a simulation script to generate the appropriate .vcd.

Note: You can use a .vcd created for power analysis to optimize your design for power during fitting by utilizing the appropriate settings in the PowerPlay power optimization list, available from Assignments > Settings > Compiler Settings > Advanced Settings (Fitter).

Using Simulation Files in Modular Design Flows

A common design practice is to create modular or hierarchical designs in which you develop each design entity separately, and then instantiate these modules in a higher-level entity to form a complete design. You can perform simulation on a complete design or on each module for verification. The PowerPlay Power Analyzer supports modular design flows when reading the signal activities from simulation files. The following figure shows an example of a modular design flow.
When specifying a simulation file (a .vcd), the software provides support to specify an associated design entity name, such that the PowerPlay Power Analyzer imports the signal activities derived from that file for the specified design entity. The PowerPlay Power Analyzer also supports the specification of multiple .vcd files for power analysis, with each having an associated design entity name to enable the integration of partial design simulations into a complete design power analysis. When specifying multiple .vcd files for your design, more than one simulation file can contain signal-activity information for the same signal.

**Note:** When you apply multiple .vcd files to the same design entity, the signal activity used in the power analysis is the equal-weight arithmetic average of each .vcd.

**Note:** When you apply multiple simulation files to design entities at different levels in your design hierarchy, the signal activity in the power analysis derives from the simulation file that applies to the most specific design entity.

The following figure shows an example of a hierarchical design. The top-level module of your design, called Top, consists of three 8b/10b decoders, followed by a mux. The software then encodes the output of the mux to produce the final output of the top-level module. An error-handling module handles any 8b/10b decoding errors. The Top module contains the top-level entity of your design and any logic not defined as part of another module. The design file for the top-level module might be a wrapper for the hierarchical entities below it, or it might contain its own logic. The following usage scenarios show common ways that you can simulate your design and import the .vcd into the PowerPlay Power Analyzer.
Complete Design Simulation

You can simulate the entire design and generate a .vcd from a third-party simulator. The PowerPlay Power Analyzer can then import the .vcd (specifying the top-level design). The resulting power analysis uses the signal activities information from the generated .vcd, including those that apply to submodules, such as decode [1-3], err1, mux1, and encode1.

Modular Design Simulation

You can independently simulate of the top-level design, and then import all the resulting .vcd files into the PowerPlay Power Analyzer. For example, you can simulate the 8b10b_dec independent of the entire design and mux, 8b10b_rxerr, and 8b10b_enc. You can then import the .vcd files generated from each simulation by specifying the appropriate instance name. For example, if the files produced by the simulations are 8b10b_dec.vcd, 8b10b_enc.vcd, 8b10b_rxerr.vcd, and mux.vcd, you can use the import specifications in the following table:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b10b_dec.vcd</td>
<td>Top</td>
</tr>
<tr>
<td>8b10b_dec.vcd</td>
<td>Top</td>
</tr>
<tr>
<td>8b10b_dec.vcd</td>
<td>Top</td>
</tr>
<tr>
<td>8b10b_rxerr.vcd</td>
<td>Top</td>
</tr>
<tr>
<td>8b10b_enc.vcd</td>
<td>Top</td>
</tr>
<tr>
<td>mux.vcd</td>
<td>Top</td>
</tr>
</tbody>
</table>

The resulting power analysis applies the simulation vectors in each file to the assigned entity. Simulation provides signal activities for the pins and for the outputs of functional blocks. If the inputs to an entity instance are input pins for the entire design, the simulation file associated with that instance does not
provide signal activities for the inputs of that instance. For example, an input to an entity such as \texttt{mux1} has its signal activity specified at the output of one of the decode entities.

**Multiple Simulations on the Same Entity**

You can perform multiple simulations of an entire design or specific modules of a design. For example, in the process of verifying the top-level design, you can have three different simulation testbenches: one for normal operation, and two for corner cases. Each of these simulations produces a separate \texttt{.vcd}. In this case, apply the different \texttt{.vcd} file names to the same top-level entity, as shown in the following table.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{normal.vcd}</td>
<td>Top</td>
</tr>
<tr>
<td>\texttt{corner1.vcd}</td>
<td>Top</td>
</tr>
<tr>
<td>\texttt{corner2.vcd}</td>
<td>Top</td>
</tr>
</tbody>
</table>

The resulting power analysis uses an arithmetic average of the signal activities calculated from each simulation file to obtain the final signal activities used. If a signal \texttt{err_out} has a toggle rate of zero transition per second in \texttt{normal.vcd}, 50 transitions per second in \texttt{corner1.vcd}, and 70 transitions per second in \texttt{corner2.vcd}, the final toggle rate in the power analysis is 40 transitions per second.

If you do not want the PowerPlay Power Analyzer to read information from multiple instances and take an arithmetic average of the signal activities, use a \texttt{.vcd} that includes only signals from the instance that you care about.

**Overlapping Simulations**

You can perform a simulation on the entire design, and more exhaustive simulations on a submodule, such as \texttt{8b10b_rxerr}. The following table lists the import specification for overlapping simulations.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{full_design.vcd}</td>
<td>Top</td>
</tr>
<tr>
<td>\texttt{error_cases.vcd}</td>
<td>Top</td>
</tr>
</tbody>
</table>

In this case, the software uses signal activities from \texttt{error_cases.vcd} for all the nodes in the generated \texttt{.vcd} and uses signal activities from \texttt{full_design.vcd} for only those nodes that do not overlap with nodes in \texttt{error_cases.vcd}. In general, the more specific hierarchy (the most bottom-level module) derives signal activities for overlapping nodes.

**Partial Simulations**

You can perform a simulation in which the entire simulation time is not applicable to signal-activity calculation. For example, if you run a simulation for 10,000 clock cycles and reset the chip for the first 2,000 clock cycles. If the PowerPlay Power Analyzer performs the signal-activity calculation over all 10,000 cycles, the toggle rates are only 80% of their steady state value (because the chip is in reset for the
first 20% of the simulation). In this case, you must specify the useful parts of the .vcd for power analysis. The **Limit VCD Period** option enables you to specify a start and end time when performing signal-activity calculations.

### Specifying Start and End Time when Performing Signal-Activity Calculations using the Limit VCD Period Option

To specify a start and end time when performing signal-activity calculations using the **Limit VCD period** option, follow these steps:

1. In the Quartus Prime software, on the Assignments menu, click **Settings**.
2. Under the Category list, click **PowerPlay Power Analyzer Settings**.
3. Turn on the **Use input file(s) to initialize toggle rates and static probabilities during power analysis** option.
4. Click **Add**.
5. In the **File name** and **Entity** fields, browse to the necessary files.
6. Under Simulation period, turn on **VCD file** and **Limit VCD period** options.
7. In the **Start time** and **End time** fields, specify the desired start and end time.
8. Click **OK**.

You can also use the following tcl or qsf assignment to specify .vcd files:

```tcl
set_global_assignment -name POWER_INPUT_FILE_NAME "test.vcd" -section_id test.vcd
set_global_assignment -name POWER_INPUT_FILE_TYPE VCD -section_id test.vcd
set_global_assignment -name POWER_VCD_FILE_START_TIME "10 ns" -section_id test.vcd
set_global_assignment -name POWER_VCD_FILE_END_TIME "1000 ns" -section_id test.vcd
set_instance_assignment -name POWER_READ_INPUT_FILE test.vcd -to test_design
```

### Related Information

- **set_power_file_assignment**
- **Add/Edit Power Input File Dialog Box**

### Node Name Matching Considerations

Node name mismatches happen when you have .vcd applied to entities other than the top-level entity. In a modular design flow, the gate-level simulation files created in different Quartus Prime projects might not match their node names with the current Quartus Prime project.

For example, you may have a file named **8b10b_enc.vcd**, which the Quartus Prime software generates in a separate project called **8b10b_enc** while simulating the 8b10b encoder. If you import the .vcd into another project called **Top**, you might encounter name mismatches when applying the .vcd to the **8b10b_enc** module in the **Top** project. This mismatch happens because the Quartus Prime software might name all the combinational nodes in the **8b10b_enc.vcd** differently than in the **Top** project.

### Glitch Filtering

The PowerPlay Power Analyzer defines a glitch as two signal transitions so closely spaced in time that the pulse, or glitch, occurs faster than the logic and routing circuitry can respond. The output of a transport
delay model simulator contains glitches for some signals. The logic and routing structures of the device form a low-pass filter that filters out glitches that are tens to hundreds of picoseconds long, depending on the device family.

Some third-party simulators use different models than the transport delay model as the default model. Different models cause differences in signal activity and power estimation. The inertial delay model, which is the ModelSim default model, filters out more glitches than the transport delay model and usually yields a lower power estimate.

**Note:** Altera recommends that you use the transport simulation model when using the Quartus Prime software glitch filtering support with third-party simulators. Simulation glitch filtering has little effect if you use the inertial simulation model.

Glitch filtering in a simulator can also filter a glitch on one logic element (LE) (or other circuit element) output from propagating to downstream circuit elements to ensure that the glitch does not affect simulated results. Glitch filtering prevents a glitch on one signal from producing non-physical glitches on all downstream logic, which can result in a signal toggle rate and a power estimate that are too high. Circuit elements in which every input transition produces an output transition, including multipliers and logic cells configured to implement XOR functions, are especially prone to glitches. Therefore, circuits with such functions can have power estimates that are too high when glitch filtering is not used.

**Note:** Altera recommends that you use the glitch filtering feature to obtain the most accurate power estimates. For .vcd files, the PowerPlay Power Analyzer flows support two levels of glitch filtering.

### Enabling First Level of Glitch Filtering

To enable the first level of glitch filtering in the Quartus Prime software for supported third-party simulators, follow these steps:

1. On the Assignments menu, click Settings.
2. In the Category list, select Simulation under EDA Tool Settings.
3. Select the Tool name to use for the simulation.
4. Turn on Enable glitch filtering.

### Enabling Second Level of Glitch Filtering

The second level of glitch filtering occurs while the PowerPlay Power Analyzer is reading the .vcd generated by a third-party simulator. To enable the second level of glitch filtering, follow these steps:

1. On the Assignments menu, click Settings.
2. In the Category list, select PowerPlay Power Analyzer Settings.
3. Under Input File(s), turn on Perform glitch filtering on VCD files.

The .vcd file reader performs filtering complementary to the filtering performed during simulation and is often not as effective. While the .vcd file reader can remove glitches on logic blocks, the file reader cannot determine how a given glitch affects downstream logic and routing, and may eliminate the impact of the glitch completely. Filtering the glitches during simulation avoids switching downstream routing and logic automatically.

**Note:** When running simulation for design verification (rather than to produce input to the PowerPlay Power Analyzer), Altera recommends that you turn off the glitch filtering option to produce the most rigorous and conservative simulation from a functionality viewpoint. When performing simulation to produce input for the PowerPlay Power Analyzer, Altera recommends that you turn on the glitch filtering to produce the most accurate power estimates.
Node and Entity Assignments

You can assign toggle rates and static probabilities to individual nodes and entities in the design. These assignments have the highest priority, overriding data from all other signal-activity sources.

You must use the Assignment Editor or Tcl commands to create the **Power Toggle Rate** and **Power Static Probability** assignments. You can specify the power toggle rate as an absolute toggle rate in transitions per second using the **Power Toggle Rate** assignment, or you can use the **Power Toggle Rate Percentage** assignment to specify a toggle rate relative to the clock domain of the assigned node for a more specific assignment made in terms of hierarchy level.

**Note:** If you use the **Power Toggle Rate Percentage** assignment, and the node does not have a clock domain, the Quartus Prime software issues a warning and ignores the assignment.

Assigning toggle rates and static probabilities to individual nodes and entities is appropriate for signals in which you have knowledge of the signal or entity being analyzed. For example, if you know that a 100 MHz data bus or memory output produces data that is essentially random (uncorrelated in time), you can directly enter a 0.5 static probability and a toggle rate of 50 million transitions per second.

The PowerPlay Power Analyzer treats bidirectional I/O pins differently. The combinational input port and the output pad for a pin share the same name. However, those ports might not share the same signal activities. For reading signal-activity assignments, the PowerPlay Power Analyzer creates a distinct name `<node_name~output>` when configuring the bidirectional signal as an output and `<node_name~result>` when configuring the signal as an input. For example, if a design has a bidirectional pin named MYPIN, assignments for the combinational input use the name MYPIN-result, and the assignments for the output pad use the name MYPIN-output.

**Note:** When you create the logic assignment in the Assignment Editor, you cannot find the MYPIN-result and MYPIN-output node names in the Node Finder. Therefore, to create the logic assignment, you must manually enter the two differentiating node names to create the assignment for the input and output port of the bidirectional pin.

**Related Information**

*Constraining Designs*

For more information about how to use the Assignment Editor in the Quartus Prime software, refer to this document.

**Timing Assignments to Clock Nodes**

For clock nodes, the PowerPlay Power Analyzer uses timing requirements to derive the toggle rate when neither simulation data nor user-entered signal-activity data is available. The **f<sub>MAX</sub>** requirements specify full cycles per second, but each cycle represents a rising transition and a falling transition. For example, a clock **f<sub>MAX</sub>** requirement of 100 MHz corresponds to 200 million transitions per second for the clock node.

**Default Toggle Rate Assignment**

You can specify a default toggle rate for primary inputs and other nodes in your design. The PowerPlay Power Analyzer uses the default toggle rate when no other method specifies the signal-activity data.

The PowerPlay Power Analyzer specifies the toggle rate in absolute terms (transitions per second), or as a fraction of the clock rate in effect for each node. The toggle rate for a clock derives from the timing settings for the clock. For example, if the PowerPlay Power Analyzer specifies a clock with an **f<sub>MAX</sub>**
constraint of 100 MHz and a default relative toggle rate of 20%, nodes in this clock domain transition in 20% of the clock periods, or 20 million transitions occur per second. In some cases, the PowerPlay Power Analyzer cannot determine the clock domain for a node because either the PowerPlay Power Analyzer cannot determine a clock domain for the node, or the clock domain is ambiguous. For example, the PowerPlay Power Analyzer may not be able to determine a clock domain for a node if the user did not specify sufficient timing assignments. In these cases, the PowerPlay Power Analyzer substitutes and reports a toggle rate of zero.

Vectorless Estimation

For some device families, the PowerPlay Power Analyzer automatically derives estimates for signal activity on nodes with no simulation or user-entered signal-activity data. Vectorless estimation statistically estimates the signal activity of a node based on the signal activities of nodes feeding that node, and on the actual logic function that the node implements. Vectorless estimation cannot derive signal activities for primary inputs. Vectorless estimation is accurate for combinational nodes, but not for registered nodes. Therefore, the PowerPlay Power Analyzer requires simulation data for at least the registered nodes and I/O nodes for accuracy.

The PowerPlay Power Analyzer Settings dialog box allows you to disable vectorless estimation. When turned on, vectorless estimation takes precedence over default toggle rates. Vectorless estimation does not override clock assignments.

To disable vectorless estimation, perform the following steps:

1. In the Quartus Prime software, on the Assignments menu, click Settings.
2. In the Category list, select PowerPlay Power Analyzer Settings.
3. Turn off the Use vectorless estimation option.

Using the PowerPlay Power Analyzer

For flows that use the PowerPlay Power Analyzer, you must first synthesize your design, and then fit it to the target device. You must either provide timing assignments for all the clocks in your design, or use a simulation-based flow to generate activity data. You must specify the I/O standard on each device input and output and the board trace model on each output in your design.

Common Analysis Flows

You can use the analysis flows in this section with the PowerPlay Power Analyzer. However, vectorless activity estimation is only available for some device families.

Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation

In the functional simulation flow, simulation provides toggle rates and static probabilities for all pins and registers in your design. Vectorless estimation fills in the values for all the combinational nodes between pins and registers, giving good results. This flow usually provides a compilation time benefit when you use the third-party RTL simulator.
RTL Simulation Limitation

RTL simulation may not provide signal activities for all registers in the post-fitting netlist because synthesis loses some register names. For example, synthesis might automatically transform state machines and counters, thus changing the names of registers in those structures.

Signal Activities from Vectorless Estimation and User-Supplied Input Pin Activities

The vectorless estimation flow provides a low level of accuracy, because vectorless estimation for registers is not entirely accurate.

Signal Activities from User Defaults Only

The user defaults only flow provides the lowest degree of accuracy.

Importance of .vcd

Altera recommends that you use a .vcd or a .saf generated by gate-level timing simulation for an accurate power estimation because gate-level timing simulation takes all the routing resources and the exact logic array resource usage into account.

Generating a .vcd

In previous versions of the Quartus Prime software, you could use either the Quartus Prime simulator or an EDA simulator to perform your simulation. The Quartus Prime software no longer supports a built-in simulator, and you must use an EDA simulator to perform simulation. Use the .vcd as the input to the PowerPlay Power Analyzer to estimate power for your design.

To create a .vcd for your design, follow these steps:

1. On the Assignments menu, click Settings.
2. In the Category list, under EDA Tool Settings, click Simulation.
3. In the Tool name list, select your preferred EDA simulator.
4. In the Format for output netlist list, select Verilog HDL, or SystemVerilog HDL, or VHDL.
5. Turn on Generate Value Change Dump (VCD) file script.

This option turns on the Map illegal HDL characters and Enable glitch filtering options. The Map illegal HDL characters option ensures that all signals have legal names and that signal toggle rates are available later in the PowerPlay Power Analyzer. The Enable glitch filtering option directs the EDA Netlist Writer to perform glitch filtering when generating VHDL Output Files, Verilog Output Files, and the corresponding Standard Delay Format Output Files for use with other EDA simulation tools. This option is available regardless of whether or not you want to generate .vcd scripts.

Note: When performing simulation using ModelSim, the +nospecify option for the vsim command disables the specify path delays and timing checks option in ModelSim. By enabling glitch filtering on the Simulation page, the simulation models include specified path delays. Thus, ModelSim might fail to simulate a design if you enabled glitch filtering and specified the +nospecify option. Altera recommends that you remove the +nospecify option from the ModelSim vsim command to ensure accurate simulation for power estimation.

6. Click Script Settings. Select the signals that you want to output to the .vcd.

With All signals selected, the generated script instructs the third-party simulator to write all connected output signals to the .vcd. With All signals except combinational lcell outputs selected, the generated
script tells the third-party simulator to write all connected output signals to the .vcd, except logic cell combinational outputs.

Note: The file can become extremely large if you write all output signals to the file because the file size depends on the number of output signals being monitored and the number of transitions that occur.

7. Click OK.
8. In the Design instance name box, type a name for your testbench.
9. Compile your design with the Quartus Prime software and generate the necessary EDA netlist and script that instructs the third-party simulator to generate a .vcd.
10. Perform a simulation with the third-party EDA simulation tool. Call the generated script in the simulation tool before running the simulation. The simulation tool generates the .vcd and places it in the project directory.

Generating a .vcd from ModelSim Software

To generate a .vcd with the ModelSim software, follow these steps:

1. In the Quartus Prime software, on the Assignments menu, click Settings.
2. In the Category list, under EDA Tool Settings, click Simulation.
3. In the Tool name list, select your preferred EDA simulator.
4. In the Format for output netlist list, select Verilog HDL, or SystemVerilog HDL, or VHDL.
5. Turn on Generate Value Change Dump (VCD) file script.
6. To generate the .vcd, perform a full compilation.
7. In the ModelSim software, compile the files necessary for simulation.
8. Load your design by clicking Start Simulation on the Tools menu, or use the vsim command.
9. Use the .vcd script created in step 6 using the following command:
   
   source <design>_dump_all_vcd_nodes.tcl

10. Run the simulation (for example, run 2000ns or run -all).
11. Quit the simulation using the quit -sim command, if required.
12. Exit the ModelSim software.
   
   If you do not exit the software, the ModelSim software might end the writing process of the .vcd improperly, resulting in a corrupt .vcd.

Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation

To successfully generate a .vcd from the full post-fit Netlist (zero delay) simulation, follow these steps:

1. Compile your design in the Quartus Prime software to generate the Netlist <project_name>.vo.
2. In <project_name>.vo, search for the include statement for <project_name>.sdo, comment the statement out, and save the file.
   
   Altera recommends that you use the Standard Delay Format Output File (.sdo) for gate-level timing simulation. The .sdo contains the delay information of each architecture primitive and routing element in your design; however, you must exclude the .sdo for zero delay simulation.
3. Generate a .vcd for power estimation by performing the steps in Generating a .vcd on page 4-17.

PowerPlay Power Analyzer Compilation Report

The following table list the items in the Compilation Report of the PowerPlay Power Analyzer section.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>The Summary section of the report shows the estimated total thermal power consumption of your design. This includes dynamic, static, and I/O thermal power consumption. The I/O thermal power includes the total I/O power drawn from the VCCIO and VCCPD power supplies and the power drawn from VCCINT in the I/O subsystem including I/O buffers and I/O registers. The report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities. For example, a <strong>Low</strong> power estimation confidence value reflects that you have provided insufficient toggle rate data, or most of the signal-activity information used for power estimation is from default or vectorless estimation settings. For more information about the input data, refer to the PowerPlay Power Analyzer Confidence Metric report.</td>
</tr>
<tr>
<td>Settings</td>
<td>The Settings section of the report shows the PowerPlay Power Analyzer settings information of your design, including the default input toggle rates, operating conditions, and other relevant setting information.</td>
</tr>
<tr>
<td>Simulation Files Read</td>
<td>The Simulation Files Read section of the report lists the simulation output file that the .vcd used for power estimation. This section also includes the file ID, file type, entity, VCD start time, VCD end time, the unknown percentage, and the toggle percentage. The unknown percentage indicates the portion of the design module unused by the simulation vectors.</td>
</tr>
<tr>
<td>Operating Conditions Used</td>
<td>The Operating Conditions Used section of the report shows device characteristics, voltages, temperature, and cooling solution, if any, during the power estimation. This section also shows the entered junction temperature or auto-computed junction temperature during the power analysis.</td>
</tr>
<tr>
<td>Thermal Power Dissipated by Block</td>
<td>The Thermal Power Dissipated by Block section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by atoms. This information provides you with estimated power consumption for each atom in your design. By default, this section does not contain any data, but you can turn on the report with the <strong>Write power dissipation by block to report file</strong> option on the PowerPlay Power Analyzer Settings page.</td>
</tr>
<tr>
<td>Thermal Power Dissipation by Block Type (Device Resource Type)</td>
<td>This Thermal Power Dissipation by Block Type (Device Resource Type) section of the report shows the estimated thermal dynamic power and thermal static power consumption categorized by block types. This information is further categorized by estimated dynamic and static power and provides an average toggle rate by block type. Thermal power is the power dissipated as heat from the FPGA device.</td>
</tr>
<tr>
<td>Thermal Power Dissipation by Hierarchy</td>
<td>This Thermal Power Dissipation by Hierarchy section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by design hierarchy. This information is further categorized by the dynamic and static power that was used by the blocks and routing in that hierarchy. This information is useful when locating modules with high power consumption in your design.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation by Clock Domain</td>
<td>The Core Dynamic Thermal Power Dissipation by Clock Domain section of the report shows the estimated total core dynamic power dissipation by each clock domain, which provides designs with estimated power consumption for each clock domain in the design. If the clock frequency for a domain is unspecified by a constraint, the clock frequency is listed as “unspecified.” For all the combinational logic, the clock domain is listed as no clock with zero MHz.</td>
</tr>
<tr>
<td>Current Drawn from Voltage Supplies</td>
<td>The Current Drawn from Voltage Supplies section of the report lists the current drawn from each voltage supply. The $V_{CCIO}$ and $V_{CCPD}$ voltage supplies are further categorized by I/O bank and by voltage. This section also lists the minimum safe power supply size (current supply ability) for each supply voltage. Minimum current requirement can be higher than user mode current requirement in cases in which the supply has a specific power up current requirement that goes beyond user mode requirement, such as the $V_{CCPD}$ power rail in Stratix III and Stratix IV devices, and the $V_{CCIO}$ power rail in Stratix IV devices. The I/O thermal power dissipation on the summary page does not correlate directly to the power drawn from the $V_{CCIO}$ and $V_{CCPD}$ voltage supplies listed in this report. This is because the I/O thermal power dissipation value also includes portions of the $V_{CCINT}$ power, such as the I/O element (IOE) registers, which are modeled as I/O power, but do not draw from the $V_{CCIO}$ and $V_{CCPD}$ supplies. The reported current drawn from the I/O Voltage Supplies (ICCIO and ICCPD) as reported in the PowerPlay Power Analyzer report includes any current drawn through the I/O into off-chip termination resistors. This can result in ICCIO and ICCPD values that are higher than the reported I/O thermal power, because this off-chip current dissipates as heat elsewhere and does not factor in the calculation of device temperature. Therefore, total I/O thermal power does not equal the sum of current drawn from each $V_{CCIO}$ and $V_{CCPD}$ supply multiplied by $V_{CCIO}$ and $V_{CCPD}$ voltage. For SoC devices or for Arria V SoC and Cyclone V SoC devices, there is no standalone ICC_AUX_SHARED current drawn information. The ICC_AUX_SHARED is reported together with ICC_AUX.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Confidence Metric Details</td>
<td>The Confidence Metric is defined in terms of the total weight of signal activity data sources for both combinational and registered signals. Each signal has two data sources allocated to it; a toggle rate source and a static probability source. The Confidence Metric Details section also indicates the quality of the signal toggle rate data to compute a power estimate. The confidence metric is low if the signal toggle rate data comes from poor predictors of real signal toggle rates in the device during an operation. Toggle rate data that comes from simulation, user-entered assignments on specific signals or entities are reliable. Toggle rate data from default toggle rates (for example, 12.5% of the clock period) or vectorless estimation are relatively inaccurate. This section gives an overall confidence rating in the toggle rate data, from low to high. This section also summarizes how many pins, registers, and combinational nodes obtained their toggle rates from each of simulation, user entry, vectorless estimation, or default toggle rate estimations. This detailed information helps you understand how to increase the confidence metric, letting you determine your own confidence in the toggle rate data.</td>
</tr>
<tr>
<td>Signal Activities</td>
<td>The Signal Activities section lists toggle rates and static probabilities assumed by power analysis for all signals with fan-out and pins. This section also lists the signal type (pin, registered, or combinational) and the data source for the toggle rate and static probability. By default, this section does not contain any data, but you can turn on the report with the Write signal activities to report file option on the PowerPlay Power Analyzer Settings page. Altera recommends that you keep the Write signal activities to report file option turned off for a large design because of the large number of signals present. You can use the Assignment Editor to specify that activities for individual nodes or entities are reported by assigning an on value to those nodes for the Power Report Signal Activities assignment.</td>
</tr>
<tr>
<td>Messages</td>
<td>The Messages section lists the messages that the Quartus Prime software generates during the analysis.</td>
</tr>
</tbody>
</table>

**Scripting Support**

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

```bash
quartus_sh --qhelp
```

**Related Information**

- [Tcl Scripting](#)
- [API Functions for Tcl](#)
Running the PowerPlay Power Analyzer from the Command–Line

The executable to run the PowerPlay Power Analyzer is `quartus_pow`. For a complete listing of all command–line options supported by `quartus_pow`, type the following command at a system command prompt:

```
quartus_pow --help
```

or:

```
quartus_sh --qhelp
```

The following lists the examples of using the `quartus_pow` executable. Type the command listed in the following section at a system command prompt. These examples assume that operations are performed on Quartus Prime project called `sample`.

To instruct the PowerPlay Power Analyzer to generate a PowerPlay EPE File:

```
quartus_pow sample --output_epe=sample.csv
```

To instruct the PowerPlay Power Analyzer to generate a PowerPlay EPE File without performing the power estimate:

```
quartus_pow sample --output_epe=sample.csv --estimate_power=off
```

To instruct the PowerPlay Power Analyzer to use a .vcd as input (sample.vcd):

```
quartus_pow sample --input_vcd=sample.vcd
```

To instruct the PowerPlay Power Analyzer to use two .vcd files as input files (sample1.vcd and sample2.vcd), perform glitch filtering on the .vcd and use a default input I/O toggle rate of 10,000 transitions per second:

```
quartus_pow sample --input_vcd=sample1.vcd --input_vcd=sample2.vcd --vcd_filter_glitches=on --default_input_io_toggle_rate=10000transitions/s
```

To instruct the PowerPlay Power Analyzer to not use an input file, a default input I/O toggle rate of 60%, no vectorless estimation, and a default toggle rate of 20% on all remaining signals:

```
quartus_pow sample --no_input_file --default_input_io_toggle_rate=60% --use_vectorless_estimation=off --default_toggle_rate=20%
```

Note: No command–line options are available to specify the information found on the PowerPlay Power Analyzer Settings Operating Conditions page. Use the Quartus Prime GUI to specify these options.

The `quartus_pow` executable creates a report file, `<revision name>.pow.rpt`. You can locate the report file in the main project directory. The report file contains the same information in PowerPlay Power Analyzer Compilation Report on page 4–18.
The following table lists the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
</tbody>
</table>
| 2014.12.15      | 14.1.0    | • Removed Signal Activities from Full Post-Fit Netlist (Timing) Simulation and Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation sections as these are no longer supported.  
  • Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Synthesis Optimizations to Compiler Settings. |
| 2014.08.18      | 14.0a10.0 | Updated "Current Drawn from Voltage Supplies" to clarify that for SoC devices or for Arria V SoC and Cyclone V SoC devices, there is no standalone ICC_AUX_SHARED current drawn information. The ICC_AUX_SHARED is reported together with ICC_AUX. |
| November 2012   | 12.1.0    | • Updated “Types of Power Analyses” on page 8–2, and “Confidence Metric Details” on page 8–23.  
  • Added “Importance of .vcd” on page 8–20, and “Avoiding Power Estimation and Hardware Measurement Mismatch” on page 8–24 |
| June 2012       | 12.0.0    | • Updated “Current Drawn from Voltage Supplies” on page 8–22.  
  • Added “Using the HPS Power Calculator” on page 8–7.                                                                                           |
| November 2011   | 10.1.1    | • Template update.  
  • Minor editorial updates.                                                                                                                                                                         |
| December 2010   | 10.1.0    | • Added links to Quartus Prime Help, removed redundant material.  
  • Moved “Creating PowerPlay EPE Spreadsheets” to page 8–6.  
  • Minor edits.                                                                                                                                                                                    |
| July 2010       | 10.0.0    | • Removed references to the Quartus Prime Simulator.  
  • Updated Table 8–1 on page 8–6, Table 8–2 on page 8–13, and Table 8–3 on page 8–14.  
  • Updated Figure 8–3 on page 8–9, Figure 8–4 on page 8–10, and Figure 8–5 on page 8–12.                                                                                                         |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| November 2009 | 9.1.0   | • Updated “Creating PowerPlay EPE Spreadsheets” on page 8–6 and “Simulation Results” on page 8–10.<br>
                             • Added “Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation” on page 8–19 and “Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation” on page 8–21.<br>
                             • Minor changes to “Generating a .vcd from ModelSim Software” on page 8–21.<br>
                             • Updated Figure 11–8 on page 11–24.                                                                 |
| March 2009    | 9.0.0   | • This chapter was chapter 11 in version 8.1.<br>
                             • Removed Figures 11-10, 11-11, 11-13, 11-14, and 11-17 from 8.1 version.                             |
| November 2008 | 8.1.0   | • Updated for the Quartus Prime software version 8.1.<br>
                             • Replaced Figure 11-3.<br>
                             • Replaced Figure 11-14.                                                                                     |
| May 2008      | 8.0.0   | • Updated Figure 11–5.<br>
                             • Updated “Types of Power Analyses” on page 11–5.<br>
                             • Updated “Operating Conditions” on page 11–9.<br>
                             • Updated “PowerPlay Power Analyzer Compilation Report” on page 11–31.<br>
                             • Updated “Current Drawn from Voltage Supplies” on page 11–32.                                              |
About Altera System Debugging Tools

The Altera® system debugging tools help you verify your FPGA designs. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. This manual provides a quick overview of the tools available in the system debugging suite and discusses the criteria for selecting the best tool for your design.

System Debugging Tools Portfolio

The Quartus Prime software provides a portfolio of system design debugging tools for real-time verification of your design. Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. The tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. The debugging logic is then compiled with your design and downloaded into the FPGA or CPLD for analysis. Because different designs can have different constraints and requirements, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device, you can choose a tool from the available debugging tools that matches the specific requirements for your design.
# System Debugging Tools Comparison

## Table 5-1: Debugging Tools Portfolio

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Typical Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Console</strong></td>
<td>Uses a Tcl interpreter to communicate with hardware modules instantiated in your design. You can use it with the Transceiver Toolkit to monitor or debug your design. System Console provides real-time in-system debugging capabilities. Using System Console, you can read from and write to Memory Mapped components in our system without the help of a processor or additional software. System Console uses Tcl as the fundamental infrastructure which means you can source scripts, set variables, write procedures, and take advantage of all the features of the Tcl scripting language.</td>
<td>You need to perform system-level debugging. For example, if you have an Avalon-MM slave or Avalon-ST interfaces, you can debug your design at a transaction level. The tool supports JTAG connectivity and TCP/IP connectivity to the target FPGA you wish to debug.</td>
</tr>
<tr>
<td><strong>Transceiver Toolkit</strong></td>
<td>The Transceiver Toolkit allows you to test and tune transceiver link signal quality. You can use a combination of bit error rate (BER), bathtub curve, and eye contour graphs as quality metrics. Auto Sweeping of physical medium attachment (PMA) settings allows you to quickly find an optimal solution.</td>
<td>You need to debug or optimize signal integrity of your board layout even before the actual design to be run on the FPGA is ready.</td>
</tr>
<tr>
<td><strong>SignalTap® II Logic Analyzer</strong></td>
<td>This logic analyzer uses FPGA resources to sample test nodes and outputs the information to the Quartus Prime software for display and analysis.</td>
<td>You have spare on-chip memory and you want functional verification of your design running in hardware.</td>
</tr>
<tr>
<td>Tool</td>
<td>Description</td>
<td>Typical Usage</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SignalProbe</strong></td>
<td>This tool incrementally routes internal signals to I/O pins while preserving results from your last place-and-routed design.</td>
<td>You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.</td>
</tr>
<tr>
<td><strong>Logic Analyzer Interface (LAI)</strong></td>
<td>This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.</td>
<td>You have limited on-chip memory, and have a large set of internal data buses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability of the tool.</td>
</tr>
<tr>
<td><strong>In-System Sources and Probes</strong></td>
<td>This tool provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.</td>
<td>You want to prototype a front panel with virtual buttons for your FPGA design.</td>
</tr>
<tr>
<td><strong>In-System Memory Content Editor</strong></td>
<td>This tool displays and allows you to edit on-chip memory.</td>
<td>You would like to view and edit the contents of on-chip memory that is not connected to a Nios II processor. You can also use the tool when you do not want to have a Nios II debug core in your system.</td>
</tr>
<tr>
<td><strong>Virtual JTAG Interface</strong></td>
<td>This megafunction allows you to communicate with the JTAG interface so that you can develop your own custom applications.</td>
<td>You have custom signals in your design that you want to be able to communicate with.</td>
</tr>
</tbody>
</table>

**Altera JTAG Interface (AJI)**

With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. System Console uses JTAG and other interfaces as well. The JTAG resource is shared among all of the on-chip debugging tools.

**Required Arbitration Logic**

For all system debugging tools except System Console, the Quartus Prime software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks, when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as

**System Debugging Tools Overview**

Altera Corporation
**Debugging Ecosystem**

To maximize debugging closure, the Quartus Prime software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general purpose debugging tools optimized for probing signals in your register transfer level (RTL) netlist. In-System Sources and Probes, the Virtual JTAG Interface, System Console, Transceiver Toolkit, and In-System Memory Content Editor, allow you to read back data from the debugging breakpoints, and to input values into your design during runtime.

Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete debugging solution.

**Figure 5-1: Debugging Ecosystem**

---

**About Analysis Tools for RTL Nodes**

The SignalTap II Logic Analyzer, SignalProbe, and LAI are designed specifically for probing and debugging RTL signals at system speed. They are general-purpose analysis tools that enable you to tap and analyze any routable node from the FPGA or CPLD. If you have spare logic and memory resources, the
SignalTap II Logic Analyzer is useful for providing fast functional verification of your design running on actual hardware.

Conversely, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the SignalProbe make it easy to view internal design signals using external equipment.

**Note:** The SignalTap II Logic Analyzer is not supported on CPLDs, because there are no memory resources available on these devices.

**Resource Usage**

The most important selection criteria for these three tools are the available resources remaining on your device after implementing your design and the number of spare pins available. You should evaluate your preferred debugging option early on in the design planning process to ensure that your board, your Quartus Prime project, and your design are all set up to support the appropriate options. Planning early can reduce time spent during debugging and eliminate the necessary late changes to accommodate your preferred debugging methodologies.

**Figure 5-2: Resource Usage per Debugging Tool**

---

**Overhead Logic**

Any debugging tool that requires the use of a JTAG connection requires the SLD infrastructure logic, for communication with the JTAG interface and arbitration between any instantiated debugging modules. This overhead logic uses around 200 logic elements (LEs), a small fraction of the resources available in any of the supported devices. The overhead logic is shared between all available debugging modules in your design. Both the SignalTap II Logic Analyzer and the LAI use a JTAG connection.

**For SignalProbe**

SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources. SignalProbe uses only routing resources to route an internal signal to a debugging test point.

**For Logic Analyzer Interface**

The LAI requires a small amount of logic to implement the multiplexing function between the signals under test, in addition to the SLD infrastructure logic. Because no data samples are stored on the chip, the LAI uses no memory resources.
For SignalTap II

The SignalTap II Logic Analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of logic resources that the SignalTap II Logic Analyzer uses is typically a small percentage of most designs. A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300 to 400 Logic Elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your SignalTap II Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the SignalTap II Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

Resource Estimation

The resource estimation feature for the SignalTap II Logic Analyzer and the LAI allows you to quickly judge if enough on-chip resources are available before compiling the tool with your design.

Figure 5-3: Resource Estimator

Pin Usage

For SignalProbe

The ratio of the number of pins used to the number of signals tapped for the SignalProbe feature is one-to-one. Because this feature can consume free pins quickly, a typical application for this feature is routing control signals to spare pins for debugging.

For Logic Analyzer Interface

The ratio of the number of pins used to the number of signals tapped for the LAI is many-to-one. It can map up to 256 signals to each debugging pin, depending on available routing resources. The control of the active signals that are mapped to the spare I/O pins is performed via the JTAG port. The LAI is ideal for routing data buses to a set of test pins for analysis.

For SignalTap II

Other than the JTAG test pins, the SignalTap II Logic Analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the SignalTap II Logic Analyzer GUI via the JTAG test port.

Usability Enhancements

The SignalTap II Logic Analyzer, SignalProbe, and LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL files. SignalProbe inserts signals directly from your post-fit database. The SignalTap II Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists.
Incremental Routing

SignalProbe uses the incremental routing feature. The incremental routing feature runs only the Fitter stage of the compilation. This leaves your compiled design untouched, except for the newly routed node or nodes. With SignalProbe, you can save as much as 90% compile time of a full compilation.

Automation Via Scripting

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the SignalTap II Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab. The System Console includes a full Tcl interpreter for scripting.

Remote Debugging

You can perform remote debugging of your system with the Quartus Prime software via the System Console. This feature allows you to debug equipment deployed in the field through an existing TCP/IP connection.

There are two Application Notes available to assist you.

- Application Note 624 describes how to set up your NIOS II system to use the System Console to perform remote debugging.
- Application Note 693 describes how to set up your Altera SoC to use the SLD tools to perform remote debugging.

Related Information

- Application Note 624: Debugging with System Console over TCP/IP
- Application Note 693: Remote Debugging over TCP/IP for Altera SoC

Suggested On-Chip Debugging Tools for Common Debugging Features

Table 5-2: Tools for Common Debugging Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>SignalProbe</th>
<th>Logic Analyzer Interface (LAI)</th>
<th>SignalTap II Logic Analyzer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Sample Depth</td>
<td>N/A</td>
<td>X</td>
<td>—</td>
<td>An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.</td>
</tr>
<tr>
<td>Feature</td>
<td>SignalProbe</td>
<td>Logic Analyzer Interface (LAI)</td>
<td>SignalTap II Logic Analyzer</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Ease in Debugging Timing Issue</td>
<td>X</td>
<td>X</td>
<td>—</td>
<td>External equipment, such as oscilloscopes and mixed signal oscilloscopes (MSOs), can be used with either LAI or SignalProbe. When used with the LAI, external equipment provides you with access to timing mode, which allows you to debug combined streams of data.</td>
</tr>
<tr>
<td>Minimal Effect on Logic Design</td>
<td>X</td>
<td>X (2)</td>
<td>X (2)</td>
<td>The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design, because it is set as a separate design partition. SignalProbe incrementally routes nodes to pins, not affecting the design at all.</td>
</tr>
<tr>
<td>Feature</td>
<td>SignalProbe</td>
<td>Logic Analyzer Interface (LAI)</td>
<td>SignalTap II Logic Analyzer</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Short Compile and Recompile Time</td>
<td>X</td>
<td>X (2)</td>
<td>X (2)</td>
<td>SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can refit their own design partitions to decrease recompilation time.</td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>N/A</td>
<td>N/A</td>
<td>X</td>
<td>The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.</td>
</tr>
<tr>
<td>I/O Usage</td>
<td>—</td>
<td>—</td>
<td>X</td>
<td>No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.</td>
</tr>
<tr>
<td>Feature</td>
<td>SignalProbe</td>
<td>Logic Analyzer Interface (LAI)</td>
<td>SignalTap II Logic Analyzer</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>N/A</td>
<td>—</td>
<td>X</td>
<td>The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but might be limited by signal integrity issues.</td>
</tr>
<tr>
<td>No JTAG Connection</td>
<td>X</td>
<td>—</td>
<td>X</td>
<td>A FPGA design with the LAI requires an active JTAG connection to a host running the Quartus Prime software. SignalProbe and SignalTap II do not require a host for debugging purposes.</td>
</tr>
<tr>
<td>Feature</td>
<td>SignalProbe</td>
<td>Logic Analyzer Interface (LAI)</td>
<td>SignalTap II Logic Analyzer</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>No External Equipment Required</td>
<td>—</td>
<td>—</td>
<td>X</td>
<td>The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus Prime software or the stand-alone SignalTap II Logic Analyzer software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.</td>
</tr>
</tbody>
</table>

Notes to Table:

1. X indicates the recommended tools for the feature.
2. — indicates that while the tool is available for that feature, that tool might not give the best results.
3. N/A indicates that the feature is not applicable for the selected tool.

About Stimulus-Capable Tools

The In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG interface enable you to use the JTAG interface as a general-purpose communication port. Though all three tools can be used to achieve the same results, there are some considerations that make one tool easier to use in certain applications than others. In-System Sources and Probes is ideal for toggling control signals. The In-System Memory Content Editor is useful for inputting large sets of test data. Finally, the Virtual JTAG interface is well suited for more advanced users who want to develop their own customized JTAG solution.

System Console provides system-level debugging at a transaction level, such as with Avalon-MM slave or Avalon-ST interfaces. You can communicate to a chip through JTAG, and TCP/IP protocols.
Console uses a Tcl interpreter to communicate with hardware modules that you have instantiated into your design.

**In-System Sources and Probes**

In-System Sources and Probes is an easy way to access JTAG resources to both read and write to your design. You can start by instantiating a megafunction into your HDL code. The megafunction contains source ports and probe ports for driving values into and sampling values from the signals that are connected to the ports, respectively. Transaction details of the JTAG interface are abstracted away by the megafunction. During runtime, a GUI displays each source and probe port by instance and allows you to read from each probe port and drive to each source port. The GUI makes this tool ideal for toggling a set of control signals during the debugging process.

**Push Button Functionality**

A good application of In-System Sources and Probes is to use the GUI as a replacement for the push buttons and LEDs used during the development phase of a project. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using `quartus_stp`. When used with the Tk toolkit, you can build your own graphical interfaces. This feature is ideal for building a virtual front panel during the prototyping phase of the design.

**In-System Memory Content Editor**

The In-System Memory Content Editor allows you to quickly view and modify memory content either through a GUI interface or through Tcl scripting commands. The In-System Memory Content Editor works by turning single-port RAM blocks into dual-port RAM blocks. One port is connected to your clock domain and data signals, and the other port is connected to the JTAG clock and data signals for editing or viewing.

**Generate Test Vectors**

Because you can modify a large set of data easily, a useful application for the In-System Memory Content Editor is to generate test vectors for your design. For example, you can instantiate a free memory block, connect the output ports to the logic under test (using the same clock as your logic under test on the system side), and create the glue logic for the address generation and control of the memory. At runtime, you can modify the contents of the memory using either a script or the In-System Memory Content Editor GUI and perform a burst transaction of the data contents in the modified RAM block synchronous to the logic being tested.

**Virtual JTAG Interface Megafunction**

The Virtual JTAG Interface megafunction provides the finest level of granularity for manipulating the JTAG resource. This megafunction allows you to build your own JTAG scan chain by exposing all of the JTAG control signals and configuring your JTAG Instruction Registers (IRs) and JTAG Data Registers (DRs). During runtime, you control the IR/DR chain through a Tcl API, or with System Console. This feature is meant for users who have a thorough understanding of the JTAG interface and want precise control over the number and type of resources used.

**System Console**

System Console is a framework that you can launch from the Quartus Prime software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access the Qsys system integration tool to perform low-level hardware debugging of your design, as well as identify a module by its path, and open and close a connection to a Qsys module. You can access your
design at a system level for purposes of loading, unloading, and transferring designs to multiple devices. Also, System Console supports the Tk toolkit for building graphical interfaces.

**Test Signal Integrity**
System Console also allows you to access commands that allow you to control how you generate test patterns, as well as verify the accuracy of data generated by test patterns. You can use JTAG debug commands in System Console to verify the functionality and signal integrity of your JTAG chain. You can test clock and reset signals.

**Board Bring-Up and Verification**
You can use System Console to access programmable logic devices on your development board, perform board bring-up, and perform verification. You can also access software running on a Nios II or Altera SoC processor, as well as access modules that produce or consume a stream of bytes.

**Test Link Signal Integrity with Transceiver Toolkit**
Transceiver Toolkit runs from the System Console framework, and allows you to run automatic tests of your transceiver links for debugging and optimizing your transceiver designs. You can use the Transceiver Toolkit GUI to set up channel links in your transceiver devices, and then automatically run EyeQ and Auto Sweep testing to view a graphical representation of your test data.

### Document Revision History

Table 5-3: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Added information that System Console supports the Tk toolkit.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Dita conversion. Added link to Remote Debugging over TCP/IP for Altera SoC Application Note.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Maintenance release. Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release</td>
</tr>
</tbody>
</table>

**Related Information**

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

System Console provides visibility into your design and allows you to perform low-level debugging on an FPGA running in real-time. System Console performs tests on debug-enabled Qsys instantiated IP cores. A variety of debug services provide read and write access to elements in your design to facilitate debugging.

You can perform the following tasks with System Console and the tools built on top of System Console:

- Bring up boards with both finalized and partially complete designs.
- Perform remote debug with internet access.
- Automate run-time verification through scripting across multiple devices in your system.
- Test serial links with point-and-click configuration tuning in the Transceiver Toolkit.
- Debug memory interfaces with the External Memory Interface Toolkit.
- Integrate your debug IP into the debug platform.
- Test the performance of your ADC and analog chain on a MAX® 10 device with the ADC Toolkit.
- Perform system verification with MATLABS/Simulink.
In the diagram below, tools are graphical interface, such as Bus Analyzer, that you use in conjunction with the System Console GUI interface.

Tcl Console is a command-line interface that provides you access to the System Console core. API supports services and enables communication with hardware, such as, Ethernet, processor, master, and bytestream services.

Some service have hardware requirements, for example the master service requires a JTAG Master or Nios II with JTAG Debug, or similar. The Bytestream service requires a JTAG UART.

**Note:** You use debug links to connect the host to the target that you are debugging.

**System Console Architecture**

![System Console Architecture Diagram](image)

**Related Information**

- External Memory Interface Documentation
- Debugging Transceiver Links Documentation on page 7-1
- Application Note 693: Remote Hardware Debugging over TCP/IP for Altera SoC
- Application Note 624: Debugging with System Console over TCP/IP
- White Paper 01208: Hardware in the Loop from the MATLAB/Simulink Environment
- System Console Online Training
Hardware Requirements for System Console

System Console runs on your host computer and communicates with your running design through debug agents. Debug agents are soft-logic embedded in some IP cores that enable debug communication with the host computer.

You instantiate debug IP cores using the Qsys IP Catalog. Some IP cores are enabled for debug by default, while others are enabled for debug with options in the parameter editor. You can use some debug agents, such as the Nios II processor with debug enabled, for multiple purposes. For example, debugging the hardware in your design, as well as debugging the code running on the Nios II processor.

When you use IP cores with embedded debug in your design, you can make large portions of the design accessible. Debug agents allow you to read and write to memory and alter peripheral registers from the host computer. For example, when you add a JTAG to Avalon Master Bridge instance to a Qsys system, you can read and write to memory-mapped slaves connected to the bridge.

Services associated with debug agents in the running design can be directly opened and closed as needed for debugging. System Console is responsible for determining and using the communication protocol with the debug agent. The communication protocol determines the best board connection to use for command and data transmission.

The Programmable SRAM Object File (.sof) provides System Console with channel communication information. When System Console opens in the Quartus Prime software or Qsys while your design is open, any existing .sof is automatically found and linked to the detected running device. You can may need to link the design and device manuallly in a complex system.

**Note:** The following IP cores in the IP Catalog do not support VHDL simulation generation in the current version of the Quartus Prime software:
- JTAG Debug Link
- SLD Hub Controller System
- USB Debug Link

**Related Information**

WP-01170 System-Level Debugging and Monitoring of FPGA Designs

IP Cores that Interact with System Console

To facilitate debugging your design with the System Console, you can include IP cores with debug interfaces that the System Console can use to send commands and receive data. By adding the appropriate debug agent to your design, System Console services can use the associated capabilities of the debug agent.

**Table 6-1: Common Services for System Console**

<table>
<thead>
<tr>
<th>Service</th>
<th>Function</th>
<th>Debug Agent Providing Service</th>
</tr>
</thead>
</table>
| master  | Access memory-mapped (Avalon-MM or AXI) slaves connected to the master interface. | • Nios II with debug  
• JTAG to Avalon Master Bridge  
• USB Debug Master |
### System Console Flow

1. Add an IP Core to your Qsys system.
2. Generate your Qsys system.
3. Compile your design in the Quartus Prime software.
4. Connect a board and program the FPGA.
5. Start System Console.
6. Locate and open a System Console service.
7. Perform debug operation(s) with the service.
8. Close the service.

### Starting System Console

In the System Console, you use Tcl scripting commands to interact with your design in both graphical and command-line interface modes. The System Console GUI panes provide design information for debugging your design.

**Related Information**

- [System Console Examples and Tutorials](#) on page 6-77
- [System Console Commands](#) on page 6-8
- [Scripting Reference Manual](#)
- [Introduction to Tcl Online Training](#)
Customizing Startup

You can customize your System Console environment, as follows:

- Adding commands to the `system_console_rc` configuration file located at:
  
  `<$HOME>/system_console/system_console_rc.tcl`

  The file in this location is the user configuration file, which only affects the owner of the home directory.

- Specifying your own design startup configuration file with the command-line argument `--rc_script=<path_to_script>`, when you launch System Console from the Nios II command shell.

- Using the `system_console_rc.tcl` file in combination with your custom `rc_script.tcl` file. In this case, the `system_console_rc.tcl` file performs System Console actions, and the `rc_script.tcl` file performs your debugging actions.

On startup, System Console automatically runs the Tcl commands in these files. The commands in the `system_console_rc.tcl` file run first, followed by the commands in the `rc_script.tcl` file.

Starting System Console from Nios II Command Shell

1. On the Windows Start menu, click All Programs > Altera > Nios II `<version>` > Nios II `<version>` > Command Shell.
2. Type `system-console`.
3. Type `--help` for System Console help.
4. Type `system-console --project_dir=<project directory>` to point to a directory that contains `.qsf` or `.sof` files.

Starting System Console Stand-Alone

You can get the stand-alone version of System Console as part of the Quartus Prime software Programmer and Tools installer on the Altera website.

1. Navigate to the Altera Download Center page and click the Additional Software tab.
2. On the Windows Start menu, click All Programs > Altera `<version>` > Programmer and Tools > System Console

Related Information

- Altera Download Center

Starting System Console from Qsys

- Click Tools > System Console.

Starting System Console from

- Click Tools > System Debugging Tools > System Console.

System Console GUI

The System Console GUI consists of a main window with multiple panes, and allows you to interact with the design currently running on the host computer.
- **System Explorer**—Displays the hierarchy of the System Console virtual file system in your design, including board connections, devices, designs, and scripts.
- **Toolkits**—Displays available toolkits including the ADC Toolkit, Transceiver Toolkit, Toolkits, GDB Server Control Panel, and Bus Analyzer. Click the **Tools** menu to launch applications.
- **Tcl Console**—A window that allows you to interact with your design using Tcl scripts, for example, sourcing scripts, writing procedures, and using System Console API.
- **Messages**—Displays status, warning, and error messages related to connections and debug actions.

Figure 6-1: System Console GUI
System Explorer Pane

The System Explorer pane displays the virtual file system for all connected debugging IP cores, and contains the following information:

- **Devices** folder—Contains information about each device connected to System Console.
- **Scripts** folder—Stores scripts for easy execution.
- **Connections** folder—Displays information about the board connections which are visible to System Console, such as USB Blaster. Multiple connections are possible.
- **Designs** folder—Displays information about project designs connected to System Console. Each design represents a .sof file that is loaded.

The Devices contains a sub-folder for each device currently connected to System Console. Each device sub-folder contains a (link) folder and sometimes contains a (files) folder. The (link) folder shows debug agents (and other hardware) that System Console can access. The (files) folder contains information about the design files loaded from the project for the device.

**Figure 6-2: System Explorer Pane**
• The figure above shows the EP4SGX230 folder under the Device folder, which contains a (link) folder. The (link) folder contains a JTAG folder, which describes the active debug connections to this device, for example, JTAG, USB, Ethernet, and agents connected to the EP4SGX230 device via a JTAG connection.

• Folders with a context menu display a context menu icon. Right-click these folders to view the context menu. For example, the Connections folder above shows a context menu icon.

• Folders that have messages display a message icon. Mouse-over these folders to view the messages. For example, the Scripts folder above a message icon.

• Debug agents that sense the clock and reset state of the target show an information or error message with a clock status icon. The icon indicates whether the clock is running (information, green), stopped (error, red), or running but in reset (error, red). For example, the trace_system_jtag_link.h2t folder above has a running clock.

**System Console Commands**

The console commands enable testing. Use console commands to identify a service by its path, and to open and close the connection. The path that identifies a service is the first argument to most System Console commands.

To initiate a service connection, do the following:

1. Identify a service by specifying its path with the get_service_paths command.
2. Open a connection to the service with the claim_service command.
3. Use Tcl and System Console commands to test the connected device.
4. Close a connection to the service with the close_service command

**Note:** For all Tcl commands, the `<format>` argument must come first.

**Table 6-2: System Console Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_service_types</td>
<td>N/A</td>
<td>Returns a list of service types that System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and design.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| get_service_paths      | • `<service-type>`  
                         | • `<device>`—Returns services in the same specified device. The argument can be a device or another service in the device.  
                         | • `<hpath>`—Returns services whose hpath starts with the specified prefix.  
                         | • `<type>`—Returns services whose debug type matches this value. Particularly useful when opening slave services.  
                         | • `<type>`—Returns services on the same development boards as the argument. Specify a board service, or any other service on the same board. | Allows you to filter the services which are returned. |
| claim_service          | • `<service-type>`  
                         | • `<service-path>`  
                         | • `<claim-group>`  
                         | • `<claims>` | Provides finer control of the portion of a service you want to use.  
                         | claim_service returns a new path which represents a use of that service. Each use is independent. Calling claim_service multiple times returns different values each time, but each allows access to the service until closed. |
| close_service          | • `<service-type>`  
                         | • `<service-path>` | Closes the specified service type at the specified path. |
| is_service_open        | • `<service-type>`  
<pre><code>                     | • `&lt;service-type&gt;` | Returns 1 if the service type provided by the path is open, 0 if the service type is closed. |
</code></pre>
<p>| get_services_to_add    | N/A | Returns a list of all services that are instantiable with the add_service command. |</p>
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_service</td>
<td>• &lt;service-type&gt;</td>
<td>Adds a service of the specified service type with the given instance name. Run <strong>get_services_to_add</strong> to retrieve a list of instantiable services. This command returns the path where the service was added. Run <strong>help add_service &lt;service-type&gt;</strong> to get specific help about that service type, including any parameters that might be required for that service.</td>
</tr>
<tr>
<td>add_service gdbserver</td>
<td>• &lt;Processor Service&gt;</td>
<td>Instantiates a gdbserver.</td>
</tr>
<tr>
<td>add_service tcp</td>
<td>• &lt;instance name&gt;</td>
<td>Allows you to connect to a TCP/IP port that provides a debug link over ethernet. See AN693 (Remote Hardware Debugging over TCP/IP for Altera SoC) for more information.</td>
</tr>
<tr>
<td>add_service transceiver_channel_rx</td>
<td>• &lt;data_pattern_checker&gt;</td>
<td>Instantiates a Transceiver Toolkit receiver channel.</td>
</tr>
<tr>
<td>add_service transceiver_channel_tx</td>
<td>• &lt;data_pattern_generator&gt;</td>
<td>Instantiates a Transceiver Toolkit transmitter channel.</td>
</tr>
<tr>
<td>add_service transceiver_debug_link</td>
<td>• &lt;transceiver_channel_tx path&gt;</td>
<td>Instantiates a Transceiver Toolkit debug link.</td>
</tr>
<tr>
<td>get_version</td>
<td>N/A</td>
<td>Returns the current System Console version and build number.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>get_claimed_services</td>
<td>• &lt;claim&gt;</td>
<td>For the given claim group, returns a list of services claimed. The returned list consists of pairs of paths and service types. Each pair is one claimed service.</td>
</tr>
<tr>
<td>refresh_connections</td>
<td>N/A</td>
<td>Scans for available hardware and updates the available service paths if there have been any changes.</td>
</tr>
<tr>
<td>send_message</td>
<td>• &lt;level&gt;</td>
<td>Sends a message of the given level to the message window. Available levels are info, warning, error, and debug.</td>
</tr>
<tr>
<td></td>
<td>• &lt;message&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Related Information**

- [Starting System Console](#) on page 6-4
- [Remote Hardware Debugging over TCP/IP for Altera SoC](#)

**Running System Console in Command-Line Mode**

You can run System Console in command line mode and work interactively or run a Tcl script. The System Console prints responses to your commands in the console window.

- `--cli`—Runs System Console in command-line mode.
- `--project_dir=<project dir>`—Directs System Console to the location of your hardware project. Also works in GUI mode.
- `--script=<your script>.tcl`—Directs System Console to run your Tcl script.
- `--help`—Lists all available commands. Typing `--help <command name>` provides the syntax and arguments of the command. System Console provides command completion if you type the beginning letters of a command and then press the Tab key.

**Locating, Opening, and Closing System Console Services**

System Console services allow you to access different parts of your running design. For example, the services you can access to memory-mapped slave interfaces. With processor services, you can access embedded processor controls. Services do not inter-mix, but a single IP core can provide multiple services. For example, the Nios II processor contains a debug core and has a memory-mapped master interface that can connect to slaves. The master service can access the memory-mapped slaves that connect to the Nios II processor. You can also use the processor service to debug your design.

**Locating Available Services**

System Console uses a virtual file system to organize the available services, which is similar to the `/dev location` on Linux systems. Board connection, device type, and IP names are all part of a service path. Instances of services are referred to by their unique service path in the file system. You can retrieve service paths for a particular service with the command `get_service_paths <service-type>`.
Example 6-1: Locating a Service Path

```tcl
# We are interested in master services.
set service_type "master"

# Get all the paths as a list.
set master_service_paths [get_service_paths $service_type]

# We are interested in the first service in the list.
set master_index 0

# The path of the first master.
set master_path [lindex $master_service_paths $master_index]

# Or condense the above statements into one statement:
set master_path [lindex [get_service_paths master] 0]
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of System Console and between versions. Use `get_service_paths` to obtain service paths rather than hard coding them into your Tcl scripts.

The string values of service paths change with different releases of the tool, so you should not infer meaning from the actual strings within the service path. Use `marker_node_info` to get information from the path.

System Console automatically discovers most services at startup. System Console automatically scans for all JTAG and USB-based service instances and retrieves their service paths. System Console does not automatically discover some services, such as TCP/IP. Use `add_service` to inform System Console about those services.

Example 6-2: Marker_node_info

You can also use the `marker_node_info` command to get information about the discovered services so you can choose the right one.

```tcl
set slave_path [get_service_paths -type altera_avalon_uart.slave slave]
array set uart_info [marker_node_info $slave_path]
echo $uart_info(full_hpath)
```

Opening and Closing Services

After you have a service path to a particular service instance, you can access the service for use.

The `claim_service` command directs System Console to start using a particular service instance, and with no additional arguments, claims a service instance for exclusive use.

Example 6-3: Opening a Service

```tcl
set service_type "master"
set claim_path [claim_service $service_type $master_path mylib]; # Claims service.
```

You can pass additional arguments to the `claim_service` command to direct System Console to start accessing a particular portion of a service instance. For example, if you use the master service to access
memory, then use `claim_service` to only access the address space between 0x0 and 0x1000. System Console then allows other users to access other memory ranges, and denies access to the claimed memory range. The `claim_service` command returns a newly created service path that you can use to access your claimed resources.

You can access a service after you open it. When you finish accessing a service instance, use the `close_service` command to direct System Console to make this resource available to other users.

**Example 6-4: Closing a Service**

```
close_service master $claim_path;  #Closes the service.
```

**System Console Services**

Altera's System Console services provide access to hardware modules instantiated in your FPGA. Services vary in the type of debug access they provide.

**SLD Service**

The SLD Service shifts values into the instruction and data registers of SLD nodes and captures the previous value. When interacting with a SLD node, start by acquiring exclusive access to the node on an opened service.

**Example 6-5: SLD Service**

```
set timeout_in_ms 1000
set lock_failed [sld_lock $sld_service_path $timeout_in_ms]
```

This code attempts to lock the selected SLD node. If it is already locked, `sld_lock` waits for the specified timeout. Confirm the procedure returns non-zero before proceeding. Set the instruction register and capture the previous one:

```
if {$lock_failed} {
    return
}
set instr 7
set delay_us 1000
set capture [sld_access_ir $sld_service_path $instr $delay_us]
```

The 1000 microsecond delay guarantees that the following SLD command executes least 1000 microseconds later. Data register access works the same way.

```
set data_bit_length 32
set delay_us 1000
set data_bytes [list 0xEF 0xBE 0xAD 0xDE]
set capture [sld_access_dr $sld_service_path $data_bit_length $delay_us \ 
$data_bytes]
```
Shift count is specified in bits, but the data content is specified as a list of bytes. The capture return value is also a list of bytes. Always unlock the SLD node once finished with the SLD service.

`sld_unlock $sld_service_path`

**Related Information**

- [SLD Commands](#) on page 6-14
- [Virtual JTAG Megafuntion documentation](#)

## SLD Commands

### Table 6-3: SLD Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sld_access_ir</code></td>
<td><code>&lt;claim-path&gt;</code></td>
<td>Shifts the instruction value into the instruction register of the specified node. Returns the previous value of the instruction. If the <code>&lt;delay&gt;</code> parameter is non-zero, then the JTAG clock is paused for this length of time after the access.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;ir-value&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;delay&gt;</code> (in µs)</td>
<td></td>
</tr>
<tr>
<td><code>sld_access_dr</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the <code>&lt;delay&gt;</code> parameter is non-zero, then the JTAG clock is paused for at least this length of time after the access. Returns the previous contents of the data register.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;size_in_bits&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;delay-in-µs&gt;</code>, <code>&lt;list_of_byte_values&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>sld_lock</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Locks the SLD chain to guarantee exclusive access.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;timeout-in-milliseconds&gt;</code></td>
<td>Returns 0 if successful. If the SLD chain is already locked by another user, tries for <code>&lt;timeout&gt;</code> ms before throwing a Tcl error. You can use the catch command if you want to handle the error.</td>
</tr>
<tr>
<td><code>sld_unlock</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Unlocks the SLD chain.</td>
</tr>
</tbody>
</table>
In-System Sources and Probes Service

The In-System Sources and Probes (ISSP) service provides scriptable access to the altsource_probe IP core in a similar manner to using the In-System Sources and Probes Editor in the Quartus Prime software.

Example 6-6: ISSP Service

Before you use the ISSP service, ensure your design works in the In-System Sources and Probes Editor. In System Console, open the service for an ISSP instance.

```bash
set issp_index 0
set issp [lindex [get_service_paths issp] 0]
set claimed_issp [claim_service issp $issp mylib]
```

View information about this particular ISSP instance.

```bash
array set instance_info [issp_get_instance_info $claimed_issp]
set source_width $instance_info(source_width)
set probe_width $instance_info(probe_width)
```

The Quartus Prime software reads probe data as a single bitstring of length equal to the probe width.

```bash
set all_probe_data [issp_read_probe_data $claimed_issp]
```

As an example, you can define the following procedure to extract an individual probe line's data.

```bash
proc get_probe_line_data {all_probe_data index} {
    set line_data [expr { ($all_probe_data >> $index) & 1 }]
    return $line_data
}
```

```bash
set initial_all_probe_data [issp_read_probe_data $claim_issp]
set initial_line_0 [get_probe_line_data $initial_all_probe_data 0]
set initial_line_5 [get_probe_line_data $initial_all_probe_data 5]
#...
set final_all_probe_data [issp_read_probe_data $claimed_issp]
set final_line_0 [get_probe_line_data $final_all_probe_data 0]
```

Similarly, the Quartus Prime software writes source data as a single bitstring of length equal to the source width.

```bash
set source_data 0xDEADBEEF
issp_write_source_data $claimed_issp $source_data
```

The currently set source data can also be retrieved.

```bash
set current_source_data [issp_read_source_data $claimed_issp]
```
As an example, you can invert the data for a 32-bit wide source by doing the following:

```tcl
set current_source_data [issp_read_source_data $claimed_issp]
set inverted_source_data [expr { $current_source_data ^ 0xFFFFFFFF }]
issp_write_source_data $claimed_issp $inverted_source_data
```

Related Information

In-System Sources and Probes Commands on page 6-16

**In-System Sources and Probes Commands**

**Note:** The valid values for ISSP claims include `read_only`, `normal`, and `exclusive`.

**Table 6-4: In-System Sources and Probes Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>issp_get_instance_info</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns a list of the configurations of the In-System Sources and Probes instance, including:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instance_index, instance_name, source_width, probe_width</td>
</tr>
<tr>
<td>issp_read_probe_data</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Retrieves the current value of the probe input. A hex string is returned representing the probe port value.</td>
</tr>
<tr>
<td>issp_read_source_data</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Retrieves the current value of the source output port. A hex string is returned representing the source port value.</td>
</tr>
<tr>
<td>issp_write_source_data</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets values for the source output port. The value can be either a hex string or a decimal value supported by the System Console Tcl interpreter.</td>
</tr>
</tbody>
</table>

**Related Information**

In-System Sources and Probes Service on page 6-15

**Monitor Service**

The monitor service builds on top of the master service to allow reads of Avalon-MM slaves at a regular interval. The service is fully software-based. The monitor service requires no extra soft-logic. This service...
streamlines the logic to do interval reads, and it offers better performance than exercising the master service manually for the reads.

**Example 6-7: Monitor Service**

Start by determining a master and a memory address range that you are interested in polling continuously.

```bash
set master_index 0
set master [lindex [get_service_paths master] $master_index]
set address 0x2000
set bytes_to_read 100
set read_interval_ms 100
```

You can use the first master to read 100 bytes starting at address 0x2000 every 100 milliseconds. Open the monitor service:

```bash
set monitor [lindex [get_service_paths monitor] 0]
set claimed_monitor [claim_service monitor $monitor mylib]
```

Notice that the master service was not opened. The monitor service opens the master service automatically. Register the previously-defined address range and time interval with the monitor service:

```bash
monitor_add_range $claimed_monitor $master $address $bytes_to_read
monitor_set_interval $claimed_monitor $read_interval_ms
```

You can add more ranges. You must define the result at each interval:

```bash
global monitor_data_buffer
set monitor_data_buffer [list]
proc store_data {monitor master address bytes_to_read} {
    global monitor_data_buffer
    set data [monitor_read_data $claimed_monitor $master $address $bytes_to_read]
    lappend monitor_data_buffer $data
}
```

The code example above, gathers the data and appends it with a global variable. `monitor_read_data` returns the range of data polled from the running design as a list. In this example, data will be a 100-element list. This list is then appended as a single element in the `monitor_data_buffer` global list. If this procedure takes longer than the interval period, the monitor service may have to skip the next one or more calls to the procedure. In this case, `monitor_read_data` will return the latest data polled. Register this callback with the opened monitor service:

```bash
set callback [list store_data $claimed_monitor $master $address $bytes_to_read]
monitor_set_callback $claimed_monitor $callback
```

Use the callback variable to call when the monitor finishes an interval. Start monitoring:

```bash
monitor_set_enabled $claimed_monitor 1
```

Immediately, the monitor reads the specified ranges from the device and invokes the callback at the specified interval. Check the contents of `monitor_data_buffer` to verify this. To turn off the monitor, use 0 instead of 1 in the above command.
Monitor Commands

You can use the Monitor commands to read many Avalon-MM slave memory locations at a regular interval.

Under normal load, the monitor service reads the data after each interval and then calls the callback. If the value you read is timing sensitive, you can use the `monitor_get_read_interval` command to read the exact time between the intervals at which the data was read.

Under heavy load, or with a callback that takes a long time to execute, the monitor service skips some callbacks. If the registers you read do not have side effects (for example, they read the total number of events since reset), skipping callbacks has no effect on your code. The `monitor_read_data` command and `monitor_get_read_interval` command are adequate for this scenario.

If the registers you read have side effects (for example, they return the number of events since the last read), you must have access to the data that was read, but for which the callback was skipped. The `monitor_read_all_data` and `monitor_get_all_read_intervals` commands provide access to this data.

Table 6-5: Main Monitoring Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor_add_range</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Adds a contiguous memory address into the monitored memory list.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;target-path&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;address&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;size&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;service path&gt;</code></td>
<td><code>&lt;service path&gt;</code> is the value returned when you opened the service.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;target-path&gt;</code></td>
<td><code>&lt;target-path&gt;</code> is returned from [lindex [get_service_paths master] n] where n is the number of the master service.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;address&gt;</code></td>
<td><code>&lt;address&gt;</code> and <code>&lt;size&gt;</code> are relative to the master service.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;size&gt;</code></td>
<td></td>
</tr>
<tr>
<td>monitor_set_callback</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Defines a Tcl expression in a single string that will be evaluated after all the memories monitored by this service are read. Typically, this expression should be specified as a Tcl procedure call with necessary argument passed in.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Tcl-expression&gt;</code></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>monitor_set_interval</td>
<td>&lt;service-path&gt;</td>
<td>Specifies the frequency of the polling action by specifying the interval</td>
</tr>
<tr>
<td></td>
<td>&lt;interval&gt;</td>
<td>between two memory reads. The actual polling frequency varies depending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on the system activity. The monitor service will try to keep it as close</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to this specification as possible.</td>
</tr>
<tr>
<td>monitor_get_interval</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current interval set which specifies the frequency of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>polling action.</td>
</tr>
<tr>
<td>monitor_set_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Enables and disables monitoring. Memory read starts after this is</td>
</tr>
<tr>
<td></td>
<td>&lt;enable(1)/disable(0)&gt;</td>
<td>enabled, and Tcl callback is evaluated after data is read.</td>
</tr>
</tbody>
</table>

Table 6-6: Monitor Callback Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor_add_range</td>
<td>&lt;service-path&gt;</td>
<td>Adds contiguous memory addresses into the monitored memory list.</td>
</tr>
<tr>
<td></td>
<td>&lt;target-path&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;address&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;size&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The &lt;target-path&gt; argument is the name of a master service to read. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address is within the address space of this service.</td>
</tr>
<tr>
<td>monitor_set_callback</td>
<td>&lt;service-path&gt;</td>
<td>Defines a Tcl expression in a single string that will be evaluated after</td>
</tr>
<tr>
<td></td>
<td>&lt;Tcl-expression&gt;</td>
<td>all the memories monitored by this service are read. Typically, this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expression should be specified as a Tcl procedure call with necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>argument passed in.</td>
</tr>
<tr>
<td>monitor_read_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of 8-bit values read from the most recent values read</td>
</tr>
<tr>
<td></td>
<td>&lt;target-path&gt;</td>
<td>from device. The memory range specified must be the same as the monitored</td>
</tr>
<tr>
<td></td>
<td>&lt;address&gt;</td>
<td>memory range as defined by monitor_add_range.</td>
</tr>
<tr>
<td></td>
<td>&lt;size&gt;</td>
<td></td>
</tr>
</tbody>
</table>
### Command| Arguments| Function
---|---|---
monitor_read_all_data| <service-path> <target-path> <address> <size>| Returns a list of 8-bit values read from all recent values read from device since last Tcl callback. The memory range specified must be within the monitored memory range as defined by monitor_add_range.

monitor_get_read_interval| <service-path> <target-path> <address> <size>| Returns the number of milliseconds between last two data reads returned by monitor_read_data.

monitor_get_all_read_intervals| <service-path> <target-path> <address> <size>| Returns a list of intervals in milliseconds between two reads within the data returned by monitor_read_all_data.

monitor_get_missing_event_count| <service-path>| Returns the number of callback events missed during the evaluation of last Tcl callback expression.

---

**Related Information**

**Monitor Service** on page 6-16

---

**Device Service**

The device service supports device-level actions.

**Example 6-8: Programming**

You can use the device service with Tcl scripting to perform device programming.

```tcl
set device_index 0 ; #Device index for target
set device [lindex [get_service_paths device] $device_index]
set sof_path [file join project_path output_files project_name.sof]
device_download_sof $device $sof_path
```

To program, all you need are the device service path and the file system path to a `.sof`. Ensure that no other service (e.g. master service) is open on the target device or else the command fails. Afterwards, you may do the following to check that the design linked to the device is the same one programmed:

```tcl
device_get_design $device
```

**Related Information**

**Device Commands** on page 6-21
Device Commands

The device commands provide access to programmable logic devices on your board. Before you use these commands, identify the path to the programmable logic device on your board using the `get_service_paths`.

Table 6-7: Device Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_download_sof</td>
<td><code>&lt;service_path&gt;</code></td>
<td>Loads the specified <code>.sof</code> to the device specified by the path.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;sof-file-path&gt;</code></td>
<td></td>
</tr>
<tr>
<td>device_get_connections</td>
<td><code>&lt;service_path&gt;</code></td>
<td>Returns all connections which go to the device at the specified path.</td>
</tr>
<tr>
<td>device_get_design</td>
<td><code>&lt;device_path&gt;</code></td>
<td>Returns the design this device is currently linked to.</td>
</tr>
</tbody>
</table>

Related Information

Device Service on page 6-20

Design Service

You can use design service commands to work with design information.

Example 6-9: Load

When you open System Console from the software or Qsys, the current project’s debug information is sourced automatically if the `.sof` has been built. In other situations, you can load manually.

```
set sof_path [file join project_dir output_files project_name.sof]
set design [design_load $sof_path]
```

System Console is now aware that this particular `.sof` has been loaded.

Example 6-10: Linking

Once a `.sof` is loaded, System Console automatically links design information to the connected device. The resultant link persists and you can choose to unlink or reuse the link on an equivalent device with the same `.sof`.

You can perform manual linking.

```
set device_index 0; # Device index for our target
set device [lindex [get_service_paths device] $device_index]
design_link $design $device
```

Manually linking fails if the target device does not match the design service. Linking fails even if the `.sof` programmed to the target is not the same as the design `.sof`. 
### Design Service Commands
Design service commands load and work with your design at a system level.

#### Table 6-8: Design Service Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>design_load</td>
<td><code>&lt;quartus-project-path&gt;</code>, <code>&lt;sof-file-path&gt;</code>, or <code>&lt;qpf-file-path&gt;</code></td>
<td>Loads a model of a design into System Console. Returns the design path. For example, if your Project File (.qpf) is in <code>c:/projects/loopback</code>, type the following command: <code>design_load {c:\projects\loopback\}</code></td>
</tr>
<tr>
<td>design_link</td>
<td><code>&lt;design-path&gt;</code> <code>&lt;device-service-path&gt;</code></td>
<td>Links a logical design with a physical device. For example, you can link a design called <code>2c35_quartus_design</code> to a 2c35 device. After you create this link, System Console creates the appropriate correspondences between the logical and physical submodules of the project.</td>
</tr>
<tr>
<td>design_extract_debug_files</td>
<td><code>&lt;design-path&gt;</code> <code>&lt;zip-file-name&gt;</code></td>
<td>Extracts debug files from a .sof to a zip file which can be emailed to Altera Support for analysis. You can specify a design path of <code>{}</code> to unlink a device and to disable auto linking for that device.</td>
</tr>
<tr>
<td>design_get_warnings</td>
<td><code>&lt;design-path&gt;</code></td>
<td>Gets the list of warnings for this design. If the design loads correctly, then an empty list returns.</td>
</tr>
</tbody>
</table>

Related Information

Design Service Commands on page 6-21
Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. You can use the bytestream service to communicate directly to the IP core that provides bytestream interfaces, such as the Altera JTAG UART or the Avalon-ST JTAG interface.

Example 6-11: Bytestream Service

The following code finds the bytestream service for your interface and opens it.

```
set bytestream_index 0
set bytestream [lindex [get_service_paths bytestream] $bytestream_index]
set claimed_bytestream [claim_service bytestream $bytestream mylib]
```

To specify the outgoing data as a list of bytes and send it through the opened service:

```
set payload [list 1 2 3 4 5 6 7 8]
bytestream_send $claimed_bytestream $payload
```

Incoming data also comes as a list of bytes.

```
set incoming_data [list]
while {[llength $incoming_data] ==0} {
  set incoming_data [bytestream_receive $claimed_bytestream 8]
}
```

Close the service when done.

```
close_service bytestream $claimed_bytestream
```

Related Information

Bytestream Commands on page 6-23

Bytestream Commands

Table 6-9: Bytestream Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytestream_send</td>
<td>&lt;service-path&gt;</td>
<td>Sends the list of bytes to the specified bytestream service. Values argument is the list of bytes to send.</td>
</tr>
<tr>
<td></td>
<td>&lt;values&gt;</td>
<td></td>
</tr>
<tr>
<td>bytestream_receive</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of bytes currently available in the specified services receive queue, up to the specified limit. Length argument is the maximum number of bytes to receive.</td>
</tr>
<tr>
<td></td>
<td>&lt;length&gt;</td>
<td></td>
</tr>
</tbody>
</table>
JTAG Debug Service

The JTAG Debug service allows you to check the state of clocks and resets within your design. The following is a JTAG Debug design flow example.

1. To identify available JTAG Debug paths:
   
   ```
   get_service_paths jtag_debug
   ```

2. To select a JTAG Debug path:
   
   ```
   set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
   ```

3. To claim a JTAG Debug service path:
   
   ```
   set claim_jtag_path [claim_service jtag_debug$jtag_debug_path mylib]
   ```

4. Running the JTAG Debug service:
   
   ```
   jtag_debug_reset_system $claim_jtag_path
   jtag_debug_loop $claim_jtag_path [list 1 2 3 4 5]
   ```

JTAG Debug Commands

JTAG Debug commands help debug the JTAG Chain connected to a device.

Table 6-10: JTAG Debug Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>jtag_debug_loop</td>
<td>&lt;service-path&gt; &lt;list_of_byte_values&gt;</td>
<td>Loops the specified list of bytes through a loopback of tdi and tdo of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. Blocks until all bytes are received. Byte values have the 0x (hexadecimal) prefix and are delineated by spaces.</td>
</tr>
<tr>
<td>jtag_debug_sample_clock</td>
<td>&lt;service-path&gt;</td>
<td>Returns the value of the clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you may need to sample the clock several times to guarantee that it is toggling.</td>
</tr>
<tr>
<td>Command</td>
<td>Argument</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>jtag_debug_sample_reset</td>
<td>&lt;service-path&gt;</td>
<td>Returns the value of the reset_n signal of the Avalon-ST JTAG Interface core. If reset_n is low (asserted), the value is 0 and if reset_n is high (deasserted), the value is 1.</td>
</tr>
<tr>
<td>jtag_debug_sense_clock</td>
<td>&lt;service-path&gt;</td>
<td>Returns the result of a sticky bit that monitors for system clock activity. If the clock has toggled since the last execution of this command, the bit is 1. Returns true if the bit has ever toggled and otherwise returns false. The sticky bit is reset to 0 on read.</td>
</tr>
<tr>
<td>jtag_debug_reset_system</td>
<td>&lt;service-path&gt;</td>
<td>Issues a reset request to the specified service. Connectivity within your device determines which part of the system is reset.</td>
</tr>
</tbody>
</table>

**Related Information**

[Verifying Clock and Reset Signals](#) on page 6-79

## Working with Toolkits

You can use Toolkit API to create tools to visualize and interact with design debug data.

Graphical widgets in the form of buttons and text fields can leverage user input to interact with debug logic. Toolkit API is the successor to the Dashboard service. Use Toolkit API with the Quartus Prime software versions 14.1 and later.

Toolkits require the following files:
- XML file that describes the plugin (.toolkit file)
- Tcl file that implements the toolkit GUI

### Registering a Toolkit

Use the `toolkit_register` command to register a System Console toolkit. You must also specify the path to the `.toolkit` file when you use the `toolkit_register` command.

When you open System Console, toolkits that appear with the `.toolkit` extension in the `$HOME/system_console/toolkits/` directory appear in the **Tools** > **Toolkits** menu.

### Opening a Toolkit

You can use the **Toolkits** tab in System Console to launch available toolkits. Each toolkit has a description, detected hardware list, and a launch button.
To launch a toolkit from the Quartus Prime software:

1. Click **Tools > System Debugging Tools > System Console**.
2. In System Console, to view available toolkits, click **Tools > Toolkits**.

You can also use following command-line command to open a System Console toolkit:

```
toolkit_open <toolkit_name>, where <toolkit_name> is the name in the .toolkit file.
```

**Note:** You can launch a toolkit in the context of a hardware resource associated with a toolkit type. If you use the command `toolkit_open <toolkit_name> <context>`, the toolkit Tcl can retrieve the `<context>` with the `set context [toolkit_get_context]` command.

### Creating a Toolkit Description File

The Toolkit Description File (.toolkit) provides the registration data for a toolkit. It does not create an instance of the toolkit GUI.

The toolkit file includes the following attributes:

- Unique `<toolkit name>.toolkit` name that describes the toolkit type.
- Internal toolkit file name.
- Toolkit display name that appears in the GUI.
- Whether the System Console **Tools > Toolkits Tools** menu displays the toolkit.
- Path to the .tcl file that implements the toolkit.
- Description of the purpose of the toolkit (optional).
- Path to an icon to display as the toolkit launcher button in System Console (optional).

**Note:** The .png format 64x64 is preferred. If the icon does not take up the whole space, then be sure that the background is transparent.

- Requirement is a toolkit that displays as associated hardware for the toolkit (optional).

**Note:** If this toolkit works with a particular type of hardware, specify the debug type name of the hardware with the `requirement` property. This enables automatic discovery of the toolkit. The debug type name of a toolkit is the name of the hw.tcl component, a dot, and the name of the interface within that component which the toolkit will use, for example, `<hw.tcl name><interface name>`.

### Related Information

- Tool kit API GUI Example .toolkit File on page 6-32
- Tool kit API GUI Example .tcl File on page 6-32

### Matching Toolkits with IP Cores

You can match any toolkit with any IP core.

- When the toolkit searches for IP, it looks for debug markers and matches IP cores to the toolkit requirements. In the toolkit file, use the requirement attribute to specify a debug type, as follows:

  ```
  <requirement><type>debug.type-name</type></requirement>
  ``

- You can create debug assignments in the hw.tcl for an IP core. hw.tcl files are available when you load the design in System Console.
- System Console discovers debug markers from identifiers in the hardware and associates with IP without direct knowledge of the design.
Toolkit API

The Toolkit API service enables you to construct GUIs for visualizing and interacting with debug data. Toolkit API is a graphical pane for the layout of your graphical widgets, which can include buttons and text fields. Widgets can pull data from other System Console services. Similarly, widgets can leverage user input to act on debug logic in your design through services.

Properties

Widget properties can push and pull information to the user interface. Widgets have properties specific to their type. For example, the button property `onClick` performs an action when the button is clicked. A label widget does not have the same property because it does not perform an action when clicked. However, both the button and label widgets have the `text` property to display text strings.

Layout

The Toolkit API service creates a widget hierarchy where the toolkit is at the top-level. The service can implement group-type widgets that contain child widgets. Layout properties dictate layout actions performed by a parent on its children.

The `expandableX` property when set as `True`, expands the widget horizontally to encompass all of the available space. The `visible` property when set as `True` allows a widget to display in the GUI.

User Input

Some widgets allow user interaction. For example, the `textField` widget is a text box that allows user entries. You access the contents of the box with the `text` property. A Tcl script can either get or set the contents of the `textField` widget with the `text` property.

Callbacks

Some widgets can perform user-specified actions, referred to as callbacks. The `textField` widget has the `onChange` property, which is called when text contents change. The `button` widget has the `onClick` property, which is called when a button is clicked. Callbacks may update widgets or interact with services based on the contents of a text field, or the state of any other widget.

Customizing Toolkit API Widgets

Use `toolkit_set_property` command to interact with the widgets that you instantiate. The `toolkit_set_property` command is most useful when you change part of the execution of a callback.

Toolkit API Script Examples

Note: To convert your Dashboard scripts to Toolkit API, do the following:

1. Add a `.toolkit` file.
2. Remove the `add_service dashboard <name of service>` command.
3. Change `dashboard_<command>` to `toolkit_<command>`.
4. Change `open_service` to `claim_service`, for example:

   Before:
   ```
   open_service slave $path
   master_read_memory $path address count
   ```

After:
```
After:

```c
set c [claim_service slave $path lib {}
master_read_memory $c address count
```

Example 6-12: Registering the Service

Toolkit API is not initialized by default. You must register the service before you can use it.

```c
toolkit_register <toolkit_file>
```

Example 6-13: Making the Toolkit Visible in System Console

Once you register a toolkit, you must explicitly make the toolkit visible. Use the `toolkit_set_property` command to modify the `visible` property of the root toolkit.

```c
toolkit_set_property $toolkit root_widget visible true
```

In this command, `$toolkit` represents the Toolkit API service. `root_widget` is the name of the root toolkit widget. `visible` is the property that allows the toolkit to be visible in System Console.

Example 6-14: Adding Widgets

Use the `toolkit_add` command to add widgets.

```c
toolkit_add my_button button "parentGroup"
```

Use the following commands to add a label widget "my_label" to the root toolkit. In the GUI, it appears as "Widget Label."

```c
set content "Text to display goes here"
toolkit_set_property $dash $name text $content
```

This command sets the `text` property to that string. In the GUI, the displayed text changes to the new value. Add one more label:

```c
toolkit_add $dash my_label_2 label self
toolkit_set_property $dash my_label_2 text "Another label"
```

Notice the new label appears to the right of the first label. Cause the layout to put the label below instead:

```c
toolkit_set_property $dash self itemsPerRow 1
```

Example 6-15: Gathering Input

Incorporate user input into your Toolkit API:

```c
set name "my_text_field"
set widget_type "textField"
```
The widget appears, but it is very small. Make the widget fill the horizontal space:

```
toolkit_set_property $dash my_text_field expandableX true
```

Now the text field is fully visible. Text can be typed into it once clicked. Type a sentence. Now, retrieve the contents of the field:

```
set content [toolkit_get_property $dash my_text_field text]
puts $content
```

This prints the contents into the console.

**Example 6-16: Updating Widgets Upon User Events**

You can make Toolkit API perform actions without interactive typing. Use callbacks to accomplish this. Start by defining a procedure that updates the first label with the text field contents:

```
proc update_my_label_with_my_text_field {dash} {
    set content [toolkit_get_property $dash my_text_field text]
    toolkit_set_property $dash my_label text $content
}
```

Run the `update_my_label_with_my_text_field $dash` command in the Tcl Console. Notice that the first label now matches the text field contents. Have the `update_my_label_with_my_text_field $dash` command called whenever the text field changes:

```
toolkit_set_property $dash my_text_field onChange "update_my_label_with_my_text_field $dash"
```

The `onChange` property is executed each time the text field changes. The effect is the first field changes to match what is typed.

**Example 6-17: Buttons**

You can use buttons to trigger actions. Create a button that changes the second label:

```
proc append_to_my_label_2 {dash suffix} {
    set old_text [toolkit_get_property $dash my_label_2 text]
    set new_text "$old_text$suffix"
    toolkit_set_property $dash my_label_2 text $new_text
}
set text_to_append ", and more"
toolkit_add $dash my_button button self
toolkit_set_property $dash my_button onClick [list append_to_my_label_2 
    $dash $text_to_append]
```

Click the button and the second label gets some text appended to it.
Example 6-18: Groups

The property itemsPerRow dictates how widgets are laid out in a group. For more complicated layouts where the number of widgets per row is different per row, use nested groups. Add a new group with more widgets per row:

```
toolkit_add $dash my_inner_group group self
toolkit_set_property $dash my_inner_group itemsPerRow 2
toolkit_add $dash inner_button_1 button my_inner_group
toolkit_add $dash inner_button_2 button my_inner_group
```

There is now a row with a group of two buttons. You can remove the border with the group name to make the nested group more seamless.

```
toolkit_set_property $dash inner_group title ""
```

The title property can be set to any other string to have the border and title text show up.

Example 6-19: Tabs

GUIs do not require all the widgets to be visible at the same time. Tabs accomplish this.

```
toolkit_add $dash my_tabs tabbedGroup self
toolkit_set_property $dash my_tabs expandableX true
toolkit_add $dash my_tab_1 group my_tabs
toolkit_add $dash my_tab_2 group my_tabs
toolkit_add $dash tabbed_label_1 label my_tab_1
toolkit_add $dash tabbed_label_2 label my_tab_2
toolkit_set_property $dash tabbed_label_1 text "in the first tab"
toolkit_set_property $dash tabbed_label_2 text "in the second tab"
```

This adds a set of two tabs, each with a group containing a label. Clicking on the tabs changes the displayed group/label.

Toolkit API GUI Example

The Toolkit API GUI example creates a button in the Toolkits pane in the System Console window, which registers the toolkit. Clicking the Launch button under Toolkit Example opens a GUI window that provides debug interaction with your design.

The Toolkit Example includes the .toolkit and .tcl files so that you can reconstruct the example on your local system.
Register Toolkit API in System Console

Toolkit API GUI Example

Analyzing and Debugging Designs with System Console

Alterra Corporation
Toolkit API GUI Example .toolkit File

The .toolkit files registers the Toolkit API to appear in the System Console.

<?xml version="1.0" encoding="UTF-8"?>
<toolkit name="toolkit_example" displayName="Toolkit Example" addMenuItem="true">
    <file> toolkit_example.tcl </file>
</toolkit>

Related Information

Creating a Toolkit Description File on page 6-26

Toolkit API GUI Example .tcl File

The Toolkit API .tcl file creates the GUI window that provides debug interaction with your design.

namespace eval Test {
    variable ledValue 0
    variable dashboardActive 0
    variable Switch_off 1

    proc toggle { position } {
        set ::Test::ledValue ${position}
    }
}
::Test::updateDashboard
}

proc sendText {} {
    set sendText [toolkit_get_property sendTextText text]
    toolkit_set_property receiveTextText text $sendText
}

proc dashBoard {} {
    if { ${::Test::dashboardActive} == 1 } {
        return -code ok "dashboard already active"
    }

    set ::Test::dashboardActive 1
    # top group widget
    #
    toolkit_add topGroup group self
    toolkit_set_property topGroup expandableX false
    toolkit_set_property topGroup expandableY false
    toolkit_set_property topGroup itemsPerRow 1
    toolkit_set_property topGroup title ""
    #
    # leds group widget
    #
    toolkit_add ledsGroup group topGroup
    toolkit_set_property ledsGroup expandableX false
    toolkit_set_property ledsGroup expandableY false
    toolkit_set_property ledsGroup itemsPerRow 2
    toolkit_set_property ledsGroup title "LED State"
    #
    # leds widgets
    #
    toolkit_add led0Button button ledsGroup
    toolkit_set_property led0Button enabled true
    toolkit_set_property led0Button expandableY false
    toolkit_set_property led0Button text "Toggle"
    toolkit_set_property led0Button onClick {::Test::toggle 1}

    toolkit_add led0LED led ledsGroup
    toolkit_set_property led0LED expandableX false
    toolkit_set_property led0LED expandableY false
    toolkit_set_property led0LED text "LED 0"
    toolkit_set_property led0LED color "green_off"

    toolkit_add led1Button button ledsGroup
    toolkit_set_property led1Button enabled true
    toolkit_set_property led1Button expandableY false
    toolkit_set_property led1Button expandableY false
    toolkit_set_property led1Button text "Turn ON"
    toolkit_set_property led1Button onClick {::Test::toggle 2}

    toolkit_add led1LED led ledsGroup
    toolkit_set_property led1LED expandableX false
    toolkit_set_property led1LED expandableY false
    toolkit_set_property led1LED text "LED 1"
    toolkit_set_property led1LED color "green_off"
# sendText widgets
#
toolkit_add sendTextGroup group topGroup
toolkit_set_property sendTextGroup expandableX false
toolkit_set_property sendTextGroup expandableY false
toolkit_set_property sendTextGroup itemsPerRow 1
toolkit_set_property sendTextGroup title "Send Data"

toolkit_add sendTextText text sendTextGroup
toolkit_set_property sendTextText expandableX false
toolkit_set_property sendTextText expandableY false
toolkit_set_property sendTextText preferredWidth 200
toolkit_set_property sendTextText preferredHeight 200
toolkit_set_property sendTextText editable true
toolkit_set_property sendTextText htmlCapable false
toolkit_set_property sendTextText text ""

toolkit_add sendTextButton button sendTextGroup
toolkit_set_property sendTextButton enabled true
toolkit_set_property sendTextButton expandableY false
toolkit_set_property sendTextButton expandableY false
toolkit_set_property sendTextButton text "Send Now"
toolkit_set_property sendTextButton onClick {::Test::sendText}

# receiveText widgets
#
toolkit_add receiveTextGroup group topGroup
toolkit_set_property receiveTextGroup expandableX false
toolkit_set_property receiveTextGroup expandableY false
toolkit_set_property receiveTextGroup itemsPerRow 1
toolkit_set_property receiveTextGroup title "Receive Data"

toolkit_add receiveTextText text receiveTextGroup
toolkit_set_property receiveTextText expandableX false
toolkit_set_property receiveTextText expandableY false
toolkit_set_property receiveTextText preferredWidth 200
toolkit_set_property receiveTextText preferredHeight 200
toolkit_set_property receiveTextText editable false
toolkit_set_property receiveTextText htmlCapable false
toolkit_set_property receiveTextText text ""

return -code ok
}
}

proc updateDashboard {} {

if { ${::Test::dashboardActive} > 0 } {

    toolkit_set_property ledsGroup title "LED State"
    if { [ expr ${::Test::ledValue} & 0x01 & ${::Test::Switch_off} ] } {
        toolkit_set_property led0LED color "green"
        set ::Test::Switch_off 0
    } else {
        toolkit_set_property led0LED color "green_off"
        set ::Test::Switch_off 1
    }
    if { [ expr ${::Test::ledValue} & 0x02 ] } {
        toolkit_set_property led1LED color "green"
    } else {
        toolkit_set_property led1LED color "green_off"
    }
}
}
Related Information
Creating a Toolkit Description File on page 6-26

Toolkit API Commands
Toolkit API commands run in the context of a unique toolkit instance.

- **toolkit_register** on page 6-36
- **toolkit_open** on page 6-37
- **get_quartus_ini** on page 6-38
- **toolkit_get_context** on page 6-39
- **toolkit_get_types** on page 6-40
- **toolkit_get_properties** on page 6-41
- **toolkit_add** on page 6-42
- **toolkit_get_property** on page 6-43
- **toolkit_set_property** on page 6-44
- **toolkit_remove** on page 6-45
- **toolkit_get_widget_dimensions** on page 6-46
toolkit_register

Description

Point to the XML file that describes the plugin (.toolkit file).

Usage

`toolkit_register <toolkit_file>`

Returns

No return value.

Arguments

`<toolkit_file>`

Path to the toolkit definition file.

Example

`toolkit_register /data/trogdor/toolkits/burninator.xml`
toolkit_open

**Description**
Opens an instance of a toolkit in System Console.

**Usage**

```
toolkit_open <toolkit_id> [<context>]
```

**Returns**
No return value.

**Arguments**

- `<toolkit_id>`
  Name of the toolkit type to open.

- `<context>`
  An optional context, such as a service path for a hardware resource that is associated with the toolkit that opens.

**Example**

```
toolkit_open my_toolkit_id
```
get_quartus_ini

Description
Returns a value from the software .ini file.

Usage
get_quartus_ini <ini> <type>

Returns
No return value.

Arguments

<ini>
Name of the software .ini setting.

$type$
(Optional) Type of .ini setting. The known types are string and enabled. If the type is enabled, the value of the .ini setting returns 1, or 0 if not enabled.

Example
set my_in_enabled [get_quartus_ini my_ini enabled]
set my_in_raw_value [get_quartus_ini my_ini]
toolkit_get_context

Description
Returns the context that was specified when the toolkit was opened. If no context was specified, returns an empty string.

Usage
toolkit_get_context

Returns
Returns the context.

Arguments
None
N/A

Example
set context [toolkit_get_context]
toolkit_get_types

Description
Returns a list of widget types.

Usage

```
toolkit_get_types
```

Returns
No return value.

Arguments
None
N/A

Example

```
set widget_names [toolkit_get_types]
```
toolkit_get_properties

Description
Returns a list of toolkit properties for a type of widget.

Usage
toolkit_get_properties <widgetType>

Returns
No return value.

Arguments
<widgetType>
Name of a type of widget.

Example
set widget_properties [toolkit_get_properties xyChart]
### toolkit_add

<table>
<thead>
<tr>
<th>Description</th>
<th>Adds a widget to the current toolkit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td><code>toolkit_add &lt;id&gt; &lt;type&gt;&lt;groupid&gt;</code></td>
</tr>
<tr>
<td>Returns</td>
<td>No return value.</td>
</tr>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>&lt;id&gt;</td>
<td>A unique ID for the widget being added.</td>
</tr>
<tr>
<td>&lt;type&gt;</td>
<td>The type of widget that is being added.</td>
</tr>
<tr>
<td>&lt;groupid&gt;</td>
<td>The ID for the parent group that will contain the new widget.</td>
</tr>
<tr>
<td>Example</td>
<td><code>toolkit_add my_button button &quot;parentGroup&quot;</code></td>
</tr>
</tbody>
</table>
toolkit_get_property

**Description**

Returns the value of a property for a specific widget.

**Usage**

`toolkit_get_property <id> <propertyName>`

**Returns**

No return value.

**Arguments**

`<id>`

A unique ID for the widget being queried.

`<tPropertyName>`

The name of the widget property being queried.

**Example**

`set enabled [toolkit_get_property my_button enabled]`
**toolkit_set_property**

**Description**
Sets the value of a property for a specific widget.

**Usage**

```
  toolkit_set_property <id><propertyName> <value>
```

**Returns**
No return value.

**Arguments**

- `<id>`
  A unique ID for the widget being modified.

- `<propertyName>`
  The name of the widget property being set.

- `<value>`
  The new value for the widget property.

**Example**

```
  toolkit_set_property my_button enabled false
```
**toolkit_remove**

**Description**
Removes a widget from the specified toolkit.

**Usage**
```
toolkit_remove <id>
```

**Returns**
No return value.

**Arguments**
`<id>`
A unique ID for the widget being removed.

**Example**
```
toolkit_remove my_button
```
**toolkit_get_widget_dimensions**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the width and height of the specified widget.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage</strong></td>
<td>toolkit_get_widget_dimensions &lt;id&gt;</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>No return value.</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>&lt;id&gt;</td>
</tr>
<tr>
<td></td>
<td>A unique ID for the widget being added.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>set dimensions [toolkit_get_widget_dimensions my_button]</td>
</tr>
</tbody>
</table>
Toolkit API Properties

*Widget Types and Properties* on page 6-48
*barChart Properties* on page 6-49
*button Properties* on page 6-50
*checkBox Properties* on page 6-51
*comboBox Properties* on page 6-52
*dial Properties* on page 6-53
*fileChooserButton Properties* on page 6-54
*group Properties* on page 6-55
*label Properties* on page 6-56
*led Properties* on page 6-57
*lineChart Properties* on page 6-58
*list Properties* on page 6-59
*pieChart Properties* on page 6-60
*tableProperties* on page 6-61
*text Properties* on page 6-62
*textField Properties* on page 6-63
*timeChart Properties* on page 6-64
*xyChart Properties* on page 6-65
### Widget Types and Properties

**Table 6-11: Toolkit API Widget Types and Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Enables or disables the widget.</td>
</tr>
<tr>
<td>expandable</td>
<td>Allows the widget to be expanded.</td>
</tr>
<tr>
<td>expandableX</td>
<td>Allows the widget to be resized horizontally if there is space available in the cell where it resides.</td>
</tr>
<tr>
<td>expandableY</td>
<td>Allows the widget to be resized vertically if there is space available in the cell where it resides.</td>
</tr>
<tr>
<td>foregroundColor</td>
<td>Sets the foreground color.</td>
</tr>
<tr>
<td>maxHeight</td>
<td>If the widget’s expandableY is set, this is the maximum height in pixels that the widget can take.</td>
</tr>
<tr>
<td>minHeight</td>
<td>If the widget’s expandableY is set, this is the minimum height in pixels that the widget can take.</td>
</tr>
<tr>
<td>maxWidth</td>
<td>If the widget’s expandableX is set, this is the maximum width in pixels that the widget can take.</td>
</tr>
<tr>
<td>minWidth</td>
<td>If the widget’s expandableX is set, this is the minimum width in pixels that the widget can take.</td>
</tr>
<tr>
<td>preferredHeight</td>
<td>The height of the widget if expandableY is not set.</td>
</tr>
<tr>
<td>preferredWidth</td>
<td>The width of the widget if expandableX is not set.</td>
</tr>
<tr>
<td>toolTip</td>
<td>Implements a mouse-over tooltip.</td>
</tr>
<tr>
<td>visible</td>
<td>Displays the widget.</td>
</tr>
</tbody>
</table>
**barChart Properties**

**Table 6-12: Toolkit API barChart Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title.</td>
</tr>
<tr>
<td>labelX</td>
<td>X-axis label text.</td>
</tr>
<tr>
<td>label</td>
<td>X-axis label text.</td>
</tr>
<tr>
<td>range</td>
<td>Y-axis value range. By default, it is auto range. Range is specified in a Tcl list, for example:</td>
</tr>
<tr>
<td></td>
<td>[list lower_numerical_value upper_numerical_value]</td>
</tr>
<tr>
<td>itemValue</td>
<td>Value is specified in a Tcl list, for example:</td>
</tr>
<tr>
<td></td>
<td>[list bar_category_str numerical_value]</td>
</tr>
</tbody>
</table>
# button Properties

## Table 6-13: Toolkit API button Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onClick</td>
<td>A Tcl command to run, usually a proc, every time the button is clicked.</td>
</tr>
<tr>
<td>text</td>
<td>The text on the button.</td>
</tr>
</tbody>
</table>
### checkBox Properties

Table 6-14: Toolkit API checkBox Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checked</td>
<td>If true, the checkbox is checked. If false, the checkbox is not checked.</td>
</tr>
<tr>
<td>onClick</td>
<td>A Tcl command to run, usually a proc, every time the checkbox is clicked.</td>
</tr>
<tr>
<td>text</td>
<td>The text on the checkbox.</td>
</tr>
</tbody>
</table>
## comboBox Properties

Table 6-15: Toolkit API comboBox Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onChange</td>
<td>A Tcl callback to run when the value of the combo box changes.</td>
</tr>
<tr>
<td>options</td>
<td>A list of options to display in the combo box.</td>
</tr>
<tr>
<td>selected</td>
<td>The index of the selected item in the combo box.</td>
</tr>
</tbody>
</table>
# dial Properties

## Table 6-16: Toolkit API dial Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>The maximum value that the dial can show.</td>
</tr>
<tr>
<td>min</td>
<td>The minimum value that the dial can show.</td>
</tr>
<tr>
<td>ticksize</td>
<td>The space between the different tick marks of the dial.</td>
</tr>
<tr>
<td>title</td>
<td>The title of the dial.</td>
</tr>
<tr>
<td>value</td>
<td>The value that the dial's needle should mark. It must be between min and max.</td>
</tr>
</tbody>
</table>
# fileChooserButton Properties

Table 6-17: Toolkit API fileChooserButton Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The text on the button.</td>
</tr>
<tr>
<td>onChoose</td>
<td>A Tcl command to run, usually a proc, every time the button is clicked.</td>
</tr>
<tr>
<td>title</td>
<td>The dialog box title.</td>
</tr>
<tr>
<td>chooserButtonText</td>
<td>chooserButtonText</td>
</tr>
</tbody>
</table>
| filter           | The file filter based on extension. Only one extension is supported. By default, all file names are allowed. The filter is specified as `[list filter_description file_extension]`, for example:  
  `[list "Text Document (.txt)" "txt"]` |
| mode             | Specifies what kind of files or directories can be selected. The default is files_only. Possible options are files_only and directories_only. |
| multiSelectionEnabled | Controls whether multiple files can be selected. False, by default.    |
| paths            | Returns a list of file paths selected in the file chooser dialog box. This property is read-only. It is most useful when used within the onClick script or a procedure when the result is freshly updated after the dialog box closes. |
### group Properties

**Table 6-18: Toolkit API group Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemsPerRow</td>
<td>The number of widgets the group can position in one row, from left to right, before moving to the next row.</td>
</tr>
<tr>
<td>title</td>
<td>The number of widgets the group can position in one row, from left to right, before moving to the next row.</td>
</tr>
</tbody>
</table>
## label Properties

### Table 6-19: Toolkit API label Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The text to show in the label.</td>
</tr>
</tbody>
</table>
## led Properties

**Table 6-20: Toolkit API led Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>The color of the LED. The options are: red_off, red, yellow_off, yellow, green_off, green, blue_off, blue, and black.</td>
</tr>
<tr>
<td>text</td>
<td>The color of the LED. The options are: red_off, red, yellow_off, yellow, green_off, green, blue_off, blue, and black.</td>
</tr>
</tbody>
</table>
## lineChart Properties

### Table 6-21: Toolkit API lineChart Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title.</td>
</tr>
<tr>
<td>labelX</td>
<td>X-axis label text.</td>
</tr>
<tr>
<td>labelY</td>
<td>Y-axis label text.</td>
</tr>
<tr>
<td>range</td>
<td>Y-axis value range. By default, it is auto range. You specify range in a Tcl list, for example:</td>
</tr>
<tr>
<td></td>
<td>[list lower_numerical_value upper_numerical_value]</td>
</tr>
</tbody>
</table>

Y-axis value range. By default, it is auto range. Range is specified in a Tcl list, for example [list lower_numerical_value upper_numerical_value].
## list Properties

### Table 6-22: Toolkit API list Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>selected</td>
<td>Index of the selected item in the combo box.</td>
</tr>
<tr>
<td>options</td>
<td>List of options to display.</td>
</tr>
<tr>
<td>onChange</td>
<td>A Tcl callback to run when the selected item in the list changes.</td>
</tr>
</tbody>
</table>
### pieChart Properties

Table 6-23: Toolkit API pieChart Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title.</td>
</tr>
<tr>
<td>itemValue</td>
<td>Item value. Value is specified in a Tcl list, for example: [list bar_category_str numerical_value]</td>
</tr>
</tbody>
</table>
### Table 6-24: Toolkit API tableProperties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnCount</td>
<td>The number of columns (Mandatory) (0, by default).</td>
</tr>
<tr>
<td>rowCount</td>
<td>The number of rows (Mandatory) (0, by default).</td>
</tr>
<tr>
<td>headerReorderingAllowed</td>
<td>Controls whether you can drag the columns (false, by default).</td>
</tr>
<tr>
<td>headerResizingAllowed</td>
<td>Controls whether you can resize all column widths. (false, by default).</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> You can resize each column individually with the columnWidthResizable property.</td>
</tr>
<tr>
<td>rowSorterEnabled</td>
<td>Controls whether you can sort the cell values in a column (false, by default).</td>
</tr>
<tr>
<td>showGrid</td>
<td>Controls whether to draw both horizontal and vertical lines (true, by default).</td>
</tr>
<tr>
<td>showHorizontalLines</td>
<td>Controls whether to draw horizontal line (true, by default).</td>
</tr>
<tr>
<td>rowIndex</td>
<td>Current row index. Zero-based. This value affects some properties below (0, by default).</td>
</tr>
<tr>
<td>columnIndex</td>
<td>Current column index. Zero-based. This value affects all column specific properties below (0, by default).</td>
</tr>
<tr>
<td>cellText</td>
<td>Specifies the text to be filled in the cell specified in the current rowIndex and columnIndex (Empty, by default).</td>
</tr>
<tr>
<td>selectedRows</td>
<td>Control or retrieve row selection.</td>
</tr>
<tr>
<td>columnHeader</td>
<td>The text to be filled in the column header.</td>
</tr>
<tr>
<td>columnHeaders</td>
<td>A list of names to define the columns for the table.</td>
</tr>
<tr>
<td>columnHorizontalAlignment</td>
<td>The cell text alignment in the specified column. Supported types are leading (default), left, center, right, trailing.</td>
</tr>
<tr>
<td>columnRowSorterType</td>
<td>The type of sorting method used. This is applicable only if rowSorterEnabled is true. Each column has its own sorting type. Supported types are string (default), int, and float.</td>
</tr>
<tr>
<td>columnWidth</td>
<td>The number of pixels used for the column width.</td>
</tr>
<tr>
<td>columnWidthResizable</td>
<td>Controls whether the column width is resizable by you (false, by default).</td>
</tr>
<tr>
<td>contents</td>
<td>The contents of the table as a list. For a table with columns A, B, and C, the format of the list will be {A1 B1 C1 A2 B2 C2 ...}.</td>
</tr>
</tbody>
</table>
## text Properties

### Table 6-25: Toolkit API text Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editable</td>
<td>Controls whether the text box is editable.</td>
</tr>
<tr>
<td>htmlCapable</td>
<td>Controls whether the text box can format HTML.</td>
</tr>
<tr>
<td>text</td>
<td>The text to show in the text box.</td>
</tr>
</tbody>
</table>
textField Properties

Table 6-26: Toolkit API textField Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editable</td>
<td>Controls whether the text box is editable.</td>
</tr>
<tr>
<td>onChange</td>
<td>A Tcl callback to run when the contents of the text box is changed.</td>
</tr>
<tr>
<td>text</td>
<td>The text to show in the text box.</td>
</tr>
</tbody>
</table>
### timeChart Properties

Table 6-27: Toolkit API timeChart Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>labelX</td>
<td>The label for the X-axis.</td>
</tr>
<tr>
<td>labelY</td>
<td>The label for the Y-axis.</td>
</tr>
<tr>
<td>latest</td>
<td>The latest value in the series.</td>
</tr>
<tr>
<td>maximumItemCount</td>
<td>The number of sample points to display in the historic record.</td>
</tr>
<tr>
<td>title</td>
<td>The title of the chart.</td>
</tr>
<tr>
<td>range</td>
<td>Sets the range for the chart. The range is of the form {low, high}. The low/high values are interpreted as doubles.</td>
</tr>
<tr>
<td>showLegend</td>
<td>Sets whether a legend for the series should be shown in the graph.</td>
</tr>
</tbody>
</table>
**xyChart Properties**

Table 6-28: Toolkit API xyChart Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title</td>
</tr>
<tr>
<td>labelX</td>
<td>X-Axis label text.</td>
</tr>
<tr>
<td>labelY</td>
<td>Y-Axis label text.</td>
</tr>
<tr>
<td>range</td>
<td>Sets the range for the chart. The range is of the form {low, high}. The low/high values are interpreted as doubles.</td>
</tr>
<tr>
<td>maximumItemCount</td>
<td>Sets the maximum number of data values to keep in a series of data. Setting this does not affect the current data, only new data added to the chart. If a series is added that has more values than the maximum count, the last-maximumItemCount number of entries will be kept.</td>
</tr>
<tr>
<td>series</td>
<td>Adds a series of data to the chart. The first value in the spec is the identifier for the series. If the same identifier is set twice, the most-recently used series data wins. If the identifier does not come with series data, that series is removed from the chart. The series is in a format such as the following: {identifier, x-1 y-1, x-2 y-2}.</td>
</tr>
<tr>
<td>showLegend</td>
<td>Sets whether a legend for the series should be shown in the graph.</td>
</tr>
</tbody>
</table>

**ADC Toolkit**

The ADC Toolkit is designed to work with MAX 10 devices and helps you understand the performance of the analog signal chain as seen by the on-board ADC hardware. The GUI displays the performance of the ADC using industry standard metrics. You can export the collected data to a .csv file and process this raw data yourself. The ADC Toolkit is built on the System Console framework and can only be operated using the GUI. There is no Tcl support for the tool.

**Prerequisites for Using the ADC Toolkit**

- Altera Modular ADC IP core
- **External Reference Voltage** if you select External in the Altera Modular ADC IP parameters
- Reference signal

The ADC Toolkit needs a sine wave signal to be fed to the analog inputs. You need the capability to precisely set the level and frequency of the reference signal. A high-precision sine wave is needed for accurate test results; however, there are useful things that can be read in **Scope** mode with any input signal.

To achieve the best testing results, the reference signal should have less distortions than the device ADC is able to resolve. If this is not the case, then you will be adding distortions from the source into the resulting ADC distortion measurements. The limiting factor is based on hardware precision.

**Note:** When applying a sine wave, the ADC should sample at 2x the fundamental sine wave frequency. There should be a low-pass filter, 3dB point set to the fundamental frequency.
Configuring the Altera Modular ADC IP Core

The Altera Modular ADC IP core needs to be included in your design. You can instantiate this IP core from the IP Catalog. When you configure this IP core in the Parameter Editor, you need to enable the Debug Path option located under Core Configuration.

There are two limitations in the software v14.1 for the Altera Modular ADC IP core. The ADC Toolkit does not support the ADC control core only option under Core Configuration. You must select a core variant that uses the standard sequencer in order for the Altera Modular ADC IP core to work with ADC Toolkit. Also, if an Avalon Master is not connected to the sequencer, you must manually start the sequencer before the ADC Toolkit will work.

Figure 6-3: Altera Modular ADC Core

Starting the ADC Toolkit

You can launch the ADC Toolkit from System Console. Before starting the ADC toolkit, you need to verify that your board is programmed. You can then load your .sof by clicking File > Load Design. If System Console was started with an active project, your design is auto-loaded when you start System Console.

There are two methods to start the ADC Toolkit. Both methods require you to have a MAX 10 device connected, programmed with a project, and linked to this project. However, the Launch command only shows up if these requirements are met. You can always start the ADC Toolkit from the Tools menu, but if the above requirements are not met, no connection will be made.

- Click Tools > ADC Toolkit
- Alternatively, click Launch from the Toolkits tab. The path for the device is displayed above the Launch button.

Note: Only one ADC Toolkit enabled device can be connected at a time.

Upon starting the ADC Toolkit, an identifier path on the ADC Toolkit tab shows you which ADC on the device is being used for this instance of the ADC Toolkit.
ADC Toolkit Flow

The ADC Toolkit GUI consists of four panels: **Frequency Selection**, **Scope**, **Signal Quality**, and **Linearity**.

1. Use the **Frequency Selection** panel to calculate the required sine wave frequency for proper signal quality testing. The ADC Toolkit will give you the nearest ideal frequency based on your desired reference signal frequency.
2. Use the **Scope** panel to tune your signal generator or inspect input signal characteristics.
3. Use the **Signal Quality** panel to test the performance of your ADC using industry standard metrics.
4. Use the **Linearity** panel to test the linearity performance of your ADC and display differential and integral non-linearity results.
Figure 6-5: ADC Toolkit GUI

Related Information

- Using the ADC Toolkit in MAX 10 Devices online training
- MAX 10 FPGA Device Overview
- MAX 10 FPGA Device Datasheet
- MAX 10 FPGA Design Guidelines
- MAX 10 Analog to Digital Converter User Guide
- Additional information about sampling frequency
  Nyquist sampling theorem and how it relates to the nominal sampling interval required to avoid aliasing.

ADC Toolkit Terms

Table 6-29: ADC Toolkit Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNR</td>
<td>The ratio of the output signal voltage level to the output noise level.</td>
</tr>
<tr>
<td>THD</td>
<td>The ratio of the sum of powers of the harmonic frequency components to the power of the fundamental/original frequency component.</td>
</tr>
<tr>
<td>SFDR</td>
<td>Characterizes the ratio between the fundamental signal and the highest spurious in the spectrum.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SINAD</td>
<td>The ratio of the RMS value of the signal amplitude to the RMS value of all other spectral components, including harmonics, but excluding DC.</td>
</tr>
<tr>
<td>ENOB</td>
<td>The number of bits with which the ADC behaves.</td>
</tr>
<tr>
<td>DNL</td>
<td>The maximum and minimum difference in the step width between actual transfer function and the perfect transfer function.</td>
</tr>
<tr>
<td>INL</td>
<td>The maximum vertical difference between the actual and the ideal curve. It indicates the amount of deviation of the actual curve from the ideal transfer curve.</td>
</tr>
</tbody>
</table>

### Setting the Frequency of the Reference Signal

You use the **Frequency Selection** panel to compute the required reference signal frequency to run the ADC performance tests. The sine wave frequency is critical and affects the validity of your test results.

**Figure 6-6: Frequency Selection Panel**

To set the frequency of the reference signal:

1. On **ADC Channel**, select the ADC channel that you plan to test. The tool populates the **Sample Size** and **Sample Frequency** fields.

2. Enter the **Desired Frequency**. This is your desired frequency for testing. You need to complete this procedure to calculate the frequency that you set your signal generator to, which will differ depending on the type of test you want to do with the ADC Toolkit.

3. Click **Calculate**.
• The closest frequency for valid testing near your desired frequency displays under both **Signal Quality Test** and **Linearity Test**.
• The nearest required sine wave frequencies are different for the signal quality test and linearity test.

4. Set your signal generator to the precise frequency given by the tool based on the type of test you want to run.

**Tuning the Signal Generator**

You use the **Scope** panel to tune your signal generator in order to achieve the best possible performance from the ADC.

**Figure 6-7: Scope Mode Panel**

To tune your signal generator:

1. **On ADC Channel**, select the ADC channel that you plan to test.
2. Enter your reference **Sample Frequency** (unless the tool can extract this value from your IP).
3. Enter your **Ref Voltage** (unless the tool can extract this value from your IP).
4. **Click Run**.
   The tool will repeatedly capture a buffer worth of data and display the data as a waveform and display additional information under **Signal Information**.
5. Tune your signal generator to use the maximum dynamic range of the ADC without clipping. Avoid hitting 0 or 4095 because your signal will likely be clipping. Look at the displayed sine wave under
Oscilloscope to see that the top and bottom peaks are evenly balanced to ensure you have selected the optimum value.

- For MAX 10 devices, you want to get as close to Min Code = 0 and Max Code = 4095 without actually hitting those values.
- The frequency should be set precisely to the value needed for testing such that coherent sampling is observed in the test window. Before moving forward, follow the suggested value for signal quality testing or linearity testing, which is displayed next to the actual frequency that is detected.
- From the Raw Data tab, you can export your data as a .csv file.

Related Information
Additional information about coherent sampling vs window sampling

Running a Signal Quality Test
The available performance metrics in signal quality test mode are the following: signal to noise ratio (SNR), total harmonic distortion (THD), spurious free dynamic range (SFDR), signal to noise and distortion ratio (SINAD), effective number of bits (ENOB), and a frequency response graph. The frequency response graph shows the signal, noise floor, and any spurs or harmonics.

The signal quality parameters are measurements relative to the carrier signal and not the full scale of the ADC.

Before you begin
Before running a signal quality test, ensure that you have set up the frequency of the reference signal using Scope mode.
To run a signal quality test:

1. On ADC Channel, select the ADC channel that you plan to test.
2. Click Run.

   From the Raw Data tab, you can export your data as a .csv file.

For signal quality tests, the signal must be coherently sampled. Based on the sampling rate and number of samples to test, specific input frequencies are required for coherent sampling.

The sample frequency for each channel is calculated based on the ADC sequencer configuration.

**Related Information**

Additional information about dynamic parameters such as SNR, THD, etc

**Running a Linearity Test**

The linearity test determines the linearity of the step sizes of each ADC code. It uses a histogram testing method which requires sinusoidal inputs which are easier to source from signal generators and DACs than other test methods.

When using Linearity test mode, your reference signal must meet specific requirements.
• The signal source covers the full code range of the ADC. Results improve if the time spent at code end is equivalent, by tuning the reference signal in **Scope** mode.

• You have to make sure if using code ends that you are not clipping the signal. Look at the signal in **Scope** mode to see that it does not look flat at the top or bottom. It may be desirable to back away from code ends and test a smaller range within the desired operating range of the ADC input signal.

• Choosing a frequency that is not an integer multiple of the sample rate and buffer size helps to ensure all code bins are filled relatively evenly to the probability density function of a sine wave. If an integer multiple is selected, some bins may be skipped entirely while others are over populated. This makes the tests results invalid. Use the frequency calculator feature to determine a good signal frequency near your desired frequency.

To run a linearity test:

1. **On ADC Channel**, select the ADC channel that you plan to test.
2. Enter the test sample size in **Burst Size**. Larger samples increase the confidence in the test results.
3. Click **Run**.

   • You can stop the test at anytime, as well as click **Run** again to continue adding to the aggregate data. To start fresh, click **Reset** after you stop a test. Anytime you change the input signal or channel, you should click **Reset** so your results are correct for a particular input.

   • There are three graphical views of the data: **Histogram** view, **DNL** view, and **INL** view.

   • From the **Raw Data** tab, you can export your data as a **.csv** file.

**ADC Toolkit Data Views**

**Histogram View**

The **Histogram** view shows how often each code appears. The graph updates every few seconds as it collects data. You can use the **Histogram** view to quickly check if your test signal is set up appropriately.
Figure 6-9: Example of Pure Sine Wave Histogram

The figure below shows the shape of a pure sine wave signal. Your reference signal should look similar.

If your reference signal is not a relatively smooth line, but has jagged edges with some bins having a value of 0, and adjacent bins with a much higher value, then the test signal frequency is not adequate. Use Scope mode to help choose a good frequency for linearity testing.
Figure 6-10: Examples of (Left) Poor Frequency Choice vs (Right) Good Frequency Choice

![Diagram showing examples of poor and good frequency choice in ADC code histograms.](image)
Differential Non-linearity View

Figure 6-11: Example of Good Differential Non-linearity

The DNL view shows the currently collected data. Ideally, you want your data to look like a straight line through the 0 on the x-axis. When there are not enough samples of data, the line appears rough. The line improves as more data is collected and averaged.

Each point in the graph represents how many LSB values a particular code differs from the ideal step size of 1 LSB. The Results box shows the highest positive and negative DNL values.
Integral Non-linearity View

Figure 6-12: Example of Good Integral Non-linearity

The INL view shows currently collected data. Ideally, with a perfect ADC and enough samples, the graph appears as a straight line through 0 on the x-axis.

Each point in the graph represents how many LSB values a particular code differs from its expected point in the voltage slope. The Results box shows the highest positive and negative INL values.

System Console Examples and Tutorials

Altera provides examples for performing board bring-up, creating a simple dashboard, and programming a Nios II processor. The System_Console.zip file contains design files for the board bring-up example. The Nios II Ethernet Standard.zip files contain the design files for the Nios II processor example.

Note: The instructions for these examples assume that you are familiar with the software, Tcl commands, and Qsys.

Related Information

On-Chip Debugging Design Examples Website
Contains the design files for the example designs that you can download.
Board Bring-Up with System Console Tutorial

You can perform low-level hardware debugging of Qsys systems with System Console. You can debug systems that include IP cores instantiated in your Qsys system or perform initial bring-up of your PCB. This board bring-up tutorial uses a Nios II Embedded Evaluation Kit (NEEK) board and USB cable. If you have a different development kit, you need to change the device and pin assignments to match your board and then recompile the design.

1. Setting Up the Board Bring-Up Design Example on page 6-78
   To load the design example into the software and program your device, follow these steps:

2. Verifying Clock and Reset Signals on page 6-79
   You can use the System Explorer pane to verify clock and reset signals.

3. Verifying Memory and Other Peripheral Interfaces on page 6-79
   The Avalon-MM service accesses memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host.

4. Qsys Modules for Board Bring-up Example on page 6-85

Related Information

- Use Cases for System Console
- Faster Board Bring-Up with System Console Demo Video

Setting Up the Board Bring-Up Design Example

To load the design example into the software and program your device, follow these steps:

1. Unzip the System_Console.zip file to your local hard drive.
2. Click File > Open Project and select Systemconsole_design_example.qpf with the software.
3. Change the device and pin assignments (LED, clock, and reset pins) in the Systemconsole_design_example.qsf file to match your board.
4. Click Processing > Start Compilation
5. To Program your device, follow these steps:
   a. Click Tools > Programmer.
   b. Click Hardware Setup.
   c. Click the Hardware Settings tab.
   d. Under Currently selected hardware, click USB-Blaster, and click Close.

   Note: If you do not see the USB-Blaster option, then your device was not detected. Verify that the USB-Blaster driver is installed, your board is powered on, and the USB cable is intact.

   This design example uses a USB-Blaster cable. If you do not have a USB-Blaster cable and you are using a different cable type, then select your cable from the Currently selected hardware options.

   e. Click Auto Detect, and then select your device.
   f. Double-click your device under File.
   g. Browse to your project folder and click Systemconsole_design_example.sof in the subdirectory output_files.
h. Turn on the **Program/Configure** option.

i. Click **Start**.

j. Close the Programmer.

6. Click **Tools > System Debugging Tools > System Console**.

**Related Information**

*System_Console.zip file*

Contains the design files for this tutorial.

---

### Verifying Clock and Reset Signals

You can use the System Explorer pane to verify clock and reset signals.

Open the appropriate node and check for either a green clock icon or a red clock icon. You can use JTAG Debug command to verify clock and reset signals.

**Related Information**

- **System Explorer Pane** on page 6-7
- **JTAG Debug Commands** on page 6-24

---

### Verifying Memory and Other Peripheral Interfaces

The Avalon-MM service accesses memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host. You can use Tcl commands to read and write to memory with a master service.

**Locating and Opening the Master Service**

```
# Select the master service type and check for available service paths.
set service_paths [get_service_paths master]

# Set the master service path.
set master_service_path [lindex $service_paths 0]

# Open the master service.
set claim_path [claim_service master $master_service_path mylib]
```

**Avalon-MM Slaves**

The **Address Map** tab shows the address range for every Qsys component. The Avalon-MM master communicates with slaves using these addresses.

The register maps for all Altera components are in their respective Data Sheets.

**Figure 6-13: Address Map**

![Address Map](image)

**Related Information**

*Data Sheets Website*
Avalon-MM Commands

Using the 8, 16, or 32 versions of the `master_read` or `master_write` commands is less efficient than using the `master_write_memory` or `master_read_memory` commands. Master commands can also be used on slave services. If you are working on a slave service, the address field can be a register (if the slave defines register names).\(^{(6)}\)

Table 6-30: Avalon-MM Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
</table>
| `master_write_memory`    | `<service-path>`
                          | `<address>`
                          | `<list_of_byte_values>`
                          | `<format>`
                          | Writes the list of byte values. Starts at the specified base address using the most efficient accesses possible with the selected master. The `<format>` argument makes this command accept data as 16 or 32-bit, instead of as bytes. For example: |
|                          |                                                                           | `master_read_memory - format 16 <service_path> <addr> <count>`                                                                         |
| `master_write_8`         | `<service-path>`
                          | `<address>`
                          | `<list_of_byte_values>`
                          | Writes the list of byte values, starting at the specified base address, using 8-bit accesses. |
| `master_write_16`        | `<service-path>`
                          | `<address>`
                          | `<list_of_16_bit_words>`
                          | Writes the list of 16-bit values, starting at the specified base address, using 16-bit accesses. |
| `master_write_from_file` | `<service-path>`
                          | `<file-name>`
                          | `<address>`
                          | Writes the entire contents of the file through the master, starting at the specified address. The file is treated as a binary file containing a stream of bytes. |
| `master_write_32`        | `<service-path>`
                          | `<address>`
                          | `<list_of_32_bit_words>`
                          | Writes the list of 32-bit values, starting at the specified base address, using 32-bit accesses. |

\(^{(6)}\) Transfers performed in 16- and 32-bit sizes are packed in little-endian format.
### Testing the PIO component

In this example design, the PIO connects to the LEDs of the board. Test if this component is operating properly and the LEDs are connected, by driving the outputs with the Avalon-MM master.

### Table 6-31: Register Map for the PIO Core

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(n-1)</td>
</tr>
<tr>
<td>0</td>
<td>data</td>
<td>R</td>
<td>Data value currently on PIO inputs.</td>
</tr>
<tr>
<td></td>
<td>read access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write access</td>
<td>W</td>
<td>New value to drive on PIO outputs.</td>
</tr>
<tr>
<td>1</td>
<td>direction</td>
<td>R/W</td>
<td>Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td>Register Name</td>
<td>R/W</td>
<td>Fields</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-----</td>
<td>--------------</td>
</tr>
<tr>
<td>2</td>
<td>interruptmask</td>
<td>R/W</td>
<td>(n-1) ... 2 1 0</td>
</tr>
<tr>
<td>3</td>
<td>edgecapture</td>
<td>R/W</td>
<td>Edge detection for each input port.</td>
</tr>
</tbody>
</table>

#Write the driver output values for the Parallel I/O component.
set offset 0x0; #Register address offset.
set value 0x7; #Only set bits 0, 1, and 2.
master_write_8 $claim_path $offset $value

#Read back the register value.
set offset 0x0
set count 0x1
master_read_8 $claim_path $offset $count

master_write_8 $claim_path 0x0 0x2; #Only set bit 1.
master_write_8 $claim_path 0x0 0xe; #Only set bits 1, 2, 3.
master_write_8 $claim_path 0x0 0x7; #Only set bits 0, 1, 2.

#Observe the LEDs turn on and off as you execute these Tcl commands.
#The LED is on if the register value is zero and off if the register value is one.
#LED 0, LED 1, and LED 2 connect to the PIO.
#LED 3 connects to the interrupt signal of the CheckSum Accelerator.

## Testing On-chip Memory

Test the memory with a recursive function that writes to incrementing memory addresses.

#Load the design example utility procedures for writing to memory.
source set_memory_values.tcl

#Write to the on-chip memory.
set base_address 0x80
set write_length 0x80
set value 0x5a5a5a5a
fill_memory $claim_path $base_address $write_length $value

#Verify the memory was written correctly.
#This utility proc returns 0 if the memory range is not uniform with this value.
verify_memory $claim_path $base_address $write_length $value

#Check that the memory is re-initialized when reset.
#Trigger reset then observe verify_memory returns 0.
set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
set claim_jtag_debug_path [claim_service jtag_debug $jtag_debug_path mylib]
jtag_debug_reset_system $claim_jtag_debug_path; #Reset the connected on-chip memory peripheral.
close_service jtag_debug $claim_jtag_debug_path
verify_memory $claim_path $base_address $write_length $value

#The on-chip memory component was parameterized to re-initialized to 0 on reset.
#Check the actual value.
master_read_8 $claim_path 0x0 0x1

## Testing the Checksum Accelerator

The Checksum Accelerator calculates the checksum of a data buffer in memory. It calculates the value for a specified memory buffer, sets the DONE bit in the status register, and asserts the interrupt signal. You
should only read the result from the controller when both the DONE bit and the interrupt signal are asserted. The host should assert the interrupt enable control bit in order to check the interrupt signal.

Table 6-32: Register Map for Checksum Component

<table>
<thead>
<tr>
<th>Offset (Bytes)</th>
<th>Hexadecimal value (after adding offset)</th>
<th>Register</th>
<th>Access</th>
<th>Bits (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31-9</td>
</tr>
<tr>
<td>0</td>
<td>0x20</td>
<td>Status</td>
<td>Read/Write to clear</td>
<td>BUSY</td>
</tr>
<tr>
<td>4</td>
<td>0x24</td>
<td>Address</td>
<td>Read/Write</td>
<td>Read Address</td>
</tr>
<tr>
<td>12</td>
<td>0x2C</td>
<td>Length</td>
<td>Read/Write</td>
<td>Length in bytes</td>
</tr>
<tr>
<td>24</td>
<td>0x38</td>
<td>Control</td>
<td>Read/Write</td>
<td>Fixed Read Address Bit</td>
</tr>
<tr>
<td>28</td>
<td>0x3C</td>
<td>Result</td>
<td>Read</td>
<td>Checksum result (upper 16 bits are zero)</td>
</tr>
</tbody>
</table>

1. #Pass the base address of the memory buffer Checksum Accelerator.
   set base_address 0x20
   set offset 4
   set address_reg [expr {($base_address + $offset)}]
   set memory_address 0x80
   master_write_32 $claim_path $address_reg $memory_address

   #Pass the memory buffer to the Checksum Accelerator.
   set length_reg [expr {($base_address + 12)}]
   set length 0x20
   master_write_32 $claim_path $length_reg $length

   #Write clear to status and control registers.
   #Status register:
   set status_reg $base_address
   master_write_32 $claim_path $status_reg 0x0
   #Control register:
   set clear 0x1
   set control_reg [expr {($base_address + 24)}]
   master_write_32 $claim_path $control_reg $clear

   #Write GO to the control register.
   set go 0x8
   master_write_32 $claim_path $control_reg $go

   #Cross check if the checksum DONE bit is set.
   master_read_32 $claim_path $status_reg 0x1

   #Is the DONE bit set?
   #If yes, check the result and you are finished with the board bring-up design example.
Testing the Checksum Accelerator

```tcl
set result_reg [expr {$base_address + 28}]
master_read_16 $claim_path $result_reg 0x1
```

2. If the result is zero and the JTAG chain works properly, the clock and reset signals work properly, and the memory works properly, then the problem is the Checksum Accelerator component.

```tcl
#Confirm if the DONE bit in the status register (bit 0) #and interrupt signal are asserted.
#Status register:
master_read_32 $claim_path $status_reg 0x1
#Check DONE bit should return a one.

#Enable interrupt and go:
set interrupt_and_go 0x18
master_write_32 $claim_path $control_reg $interrupt_and_go
```

3. Check the control enable to see the interrupt signal. LED 3 (MSB) should be off. This indicates the interrupt signal is asserted.

4. You have narrowed down the problem to the data path. View the RTL to check the data path.

5. Open the `Checksum_transform.v` file from your project folder.

   ```
   <unzip dir>/System_Console/ip/checksum_accelerator/checksum_accelerator.v
   ```

6. Notice that the `data_out` signal is grounded in Figure 6-14 (uncommented line 87 and comment line 88). Fix the problem.

7. Save the file and regenerate the Qsys system.

8. Re-compile the design and reprogram your device.

9. Redo the above steps, starting with Verifying Memory and Other Peripheral Interfaces on page 6-79 or run the Tcl script included with this design example.

```tcl
source set_memory_and_run_checksum.tcl
```
The Qsys design for this example includes the following modules:

- **JTAG to Avalon Master Bridge**—Provides System Console host access to the memory-mapped IP in the design via the JTAG interface.
- **On-chip memory**—Simplest type of memory for use in an FPGA-based embedded system. The memory is implemented on the FPGA; consequently, external connections on the circuit board are not necessary.
- **Parallel I/O (PIO) module**—Provides a memory-mapped interface for sampling and driving general I/O ports.
- **Checksum Accelerator**—Calculates the checksum of a data buffer in memory. The Checksum Accelerator consists of the following:
  - Checksum Calculator ([checksum_transform.v](#))
  - Read Master ([slave.v](#))
  - Checksum Controller ([latency_aware_read_master.v](#))

**Checksum Accelerator Functionality**

The base address of the memory buffer and data length passes to the Checksum Controller from a memory-mapped master. The Read Master continuously reads data from memory and passes the data to the Checksum Calculator. When the checksum calculations finish, the Checksum Calculator issues a valid signal along with the checksum result to the Checksum Controller. The Checksum Controller sets the

---

**Figure 6-14: Checksum.v File**

```vhdl
// First folding
assign first_folded_num = initial_num[32] + initial_num[16:14] + initial_num[15:0]; // this result is at most 17 bits wide (16 bits with rollover)

// Second folding and optional inversion, this result is at most 16 bits wide
assign data_out = (invert == 1) ? (first_folded_num[16] + first_folded_num[15:0]) : (first_folded_num[18] + first_folded_num[15:0]);

endmodule
```
DONE bit in the status register and also asserts the interrupt signal. You should only read the result from
the Checksum Controller when the DONE bit and interrupt signal are asserted.

Nios II Processor Example

This example programs the Nios II processor on your board to run the count binary software example
included in the Nios II installation. This is a simple program that uses an 8-bit variable to repeatedly
count from 0x00 to 0xFF. The output of this variable is displayed on the LEDs on your board. After
programming the Nios II processor, you use System Console processor commands to start and stop the
processor.

To run this example, perform the following steps:

1. Download the Nios II Ethernet Standard Design Example for your board from the Altera website.
2. Create a folder to extract the design. For this example, use C:\Count_binary.
3. Unzip the Nios II Ethernet Standard Design Example into C:\Count_binary.
4. In a Nios II command shell, change to the directory of your new project.
5. Program your board. In a Nios II command shell, type the following:

   nios2-configure-sof niosii_ethernet_standard_<board_version>.sof

6. Using Nios II Software Build Tools for Eclipse, create a new Nios II Application and BSP from
   Template using the Count Binary template and targeting the Nios II Ethernet Standard Design
   Example.
7. To build the executable and linkable format (ELF) file (.elf) for this application, right-click the Count
   Binary project and select Build Project.
8. Download the .elf file to your board by right-clicking Count Binary project and selecting Run As,
   Nios II Hardware.
   - The LEDs on your board provide a new light show.
9. Type the following:

   system-console; #Start System Console.
   #Set the processor service path to the Nios II processor.
   set niosii_proc [lindex [get_service_paths processor] 0]
   set claimed_proc [claim_service processor $niosii_proc mylib]; #Open the service.
   processor_stop $claimed_proc; #Stop the processor.
   #The LEDs on your board freeze.
   processor_run $claimed_proc; #Start the processor.
   #The LEDs on your board resume their previous activity.
   processor_stop $claimed_proc; #Stop the processor.
   close_service processor $claimed_proc; #Close the service.

   - The processor_step, processor_set_register, and processor_get_register commands
     provide additional control over the Nios II processor.

Related Information

- Processor Commands on page 6-87
- Nios II Ethernet Standard Design Example
- Nios II Software Build Tools User Guide
## Processor Commands

### Table 6-33: Processor Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor_download_elf</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Downloads the given Executable and Linking Format File (.elf) to memory using the master service associated with the processor. Sets the processor's program counter to the .elf entry point.</td>
</tr>
<tr>
<td>processor_in_debug_mode</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns a non-zero value if the processor is in debug mode.</td>
</tr>
<tr>
<td>processor_reset</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Resets the processor and places it in debug mode.</td>
</tr>
<tr>
<td>processor_run</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Puts the processor into run mode.</td>
</tr>
<tr>
<td>processor_stop</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Puts the processor into debug mode.</td>
</tr>
<tr>
<td>processor_step</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Executes one assembly instruction.</td>
</tr>
<tr>
<td>processor_get_register_names</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns a list with the names of all of the processor's accessible registers.</td>
</tr>
<tr>
<td>processor_get_register</td>
<td><code>&lt;service-path&gt;</code></td>
<td>&lt;register_name&gt;</td>
</tr>
<tr>
<td>processor_set_register</td>
<td><code>&lt;service-path&gt;</code></td>
<td>&lt;register_name&gt; &lt;value&gt;</td>
</tr>
</tbody>
</table>

### Related Information

**Nios II Processor Example** on page 6-86

---

(7) If your system includes a Nios II/f core with a data cache, it may complicate the debugging process. If you suspect the Nios II/f core writes to memory from the data cache at nondeterministic intervals; thereby, overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.
On-Board USB Blaster II Support

System Console supports an On-Board USB-Blaster™ II circuit via the USB Debug Master IP component. This IP core supports the master service.

Not all Stratix V boards support the On-Board USB-Blaster II. For example, the transceiver signal integrity board does not support the On-Board USB-Blaster II.

About Using MATLAB and Simulink in a System Verification Flow

System Console can be used with MATLAB and Simulink to perform system development testing. You can use the Altera Hardware in the Loop (HIL) tools to set up a system verification flow. In this approach, the design is deployed to hardware and runs in real time. The surrounding components in your system are simulated in a software environment. The HIL approach allows you to use the flexibility of software tools with the real-world accuracy and speed of hardware. You can gradually introduce more hardware components to your system verification testbench. This gives you more control over the integration process as you tune and validate your system. When your full system is integrated, the HIL approach allows you to provide stimuli via software to test your system under a variety of scenarios.

Advantages of HIL Approach

- Avoid long computational delays for algorithms with high processing rates
- API helps to control, debug, visualize, and verify FPGA designs all within the MATLAB environment
- FPGA results are read back by the MATLAB software for further analysis and display

Required Tools and Components

- MATLAB software
- DSP Builder software
- software
- Altera FPGA

**Note:** The System Console MATLAB API is included in the DSP Builder installation bundle.
Supported MATLAB API Commands

You can perform your work from the MATLAB environment and leverage the capability of System Console to read and write to masters and slaves. By using the supported MATLAB API commands, you do not have to launch the System Console software. The supported commands are the following:

- `SystemConsole.refreshMasters;`
- `M = SystemConsole.openMaster(1);`
- `M.write(type, byte address, data);`
- `M.read(type, byte address, number of words);`
- `M.close`

Example 6-20: MATLAB API Script Example

```matlab
SystemConsole.refreshMasters; %Investigate available targets
M = SystemConsole.openMaster(1); %Creates connection with FPGA target

%%%%%% User Application %.%%%%%%%

....
M.write('uint32',write_address,data); %Send data to FPGA target
....
data = M.read('uint32',read_address,size); %Read data from FPGA target
....

%%%%%%% End of User Application %%%%%%%%
M.close; %Terminates connection to FPGA target
High-Level Flow

1. Install the DSP Builder software so you have the necessary libraries to enable this flow
2. Build your design using Simulink and the DSP Builder libraries (DSP Builder helps to convert the Simulink design to HDL)
3. Include Avalon-MM components in your design (DSP Builder can port non-Avalon-MM components)
4. Include Signals and Control blocks in your design
5. Use boundary blocks to separate synthesizable and non-synthesizable logic
6. Integrate your DSP system in Qsys
7. Program your Altera FPGA
8. Use the supported MATLAB API commands to interact with your Altera FPGA

Related Information

- Hardware in the Loop from the MATLAB/Simulink Environment white paper
- System in the Loop - Enabling Real-Time FPGA Verification within MATLAB website
- DSP Builder website

Deprecated Commands

The table lists commands that have been deprecated. These commands are currently supported, but are targeted for removal from System Console.

Note: All dashboard_<name> commands are deprecated and replaced with toolkit_<name> commands for Quartus Prime software 15.1, and later.

Table 6-34: Deprecated Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_service</td>
<td>&lt;service_type&gt; &lt;service_path&gt;</td>
<td>Opens the specified service type at the specified path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Calls to open_service may be replaced with calls to claim_service providing that the return value from claim_service is stored and used to access and close the service.</td>
</tr>
</tbody>
</table>
Table 6-35: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Edits to Toolkit API content and command format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Toolkit API design example.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added graphic to <em>Introduction to System Console</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deprecated Dashboard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Edits to Toolkit API content and command format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added Toolkit API design example.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added graphic to <em>Introduction to System Console</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deprecated Dashboard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>October 2015</td>
<td>15.1.0</td>
<td>• Added content for Toolkit API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Required .toolkit and Tcl files</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Registering and launching the toolkit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Toolkit discovery and matching toolkits to IP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Toolkit API commands table</td>
</tr>
<tr>
<td>May 2015</td>
<td>15.0.0</td>
<td>Added information about how to download and start System Console stand-alone.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• Added overview and procedures for using ADC Toolkit on MAX 10 devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added overview for using MATLABS/Simulink Environment with System Console for system verification.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated design examples for the following: board bring-up, dashboard service, Nios II processor, design service, device service, monitor service, bytestream service, SLD service, and ISSP service.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Re-organization of sections. Added high-level information with block diagram, workflow, SLD overview, use cases, and example Tcl scripts.</td>
</tr>
<tr>
<td>June 2013</td>
<td>13.0.0</td>
<td>Updated Tcl command tables. Added board bring-up design example. Removed SOPC Builder content.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>Re-organization of content.</td>
</tr>
<tr>
<td>August 2012</td>
<td>12.0.1</td>
<td>Moved Transceiver Toolkit commands to Transceiver Toolkit chapter.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Maintenance release. This chapter adds new System Console features.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>Maintenance release. This chapter adds new System Console features.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Maintenance release. This chapter adds new System Console features.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Maintenance release. This chapter adds new commands and references for Qsys.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Handbook*, refer to the Quartus Handbook Archive.
The Transceiver Toolkit helps you to optimize high-speed serial links in your board design. The Transceiver Toolkit provides real-time control, monitoring, and debugging of the transceiver links running on your board.

Once you correctly configure a debugging system, you can control the transmitter or receiver channels to optimize transceiver settings and hardware features. The toolkit tests bit-error rate (BER) while running multiple links at target data rate. Run auto sweep tests to identify the best physical media attachment (PMA) settings for each link. EyeQ graphs display the receiver horizontal and vertical eye margin during testing. The toolkit supports testing of multiple devices across multiple boards simultaneously.

Figure 7-1: Transceiver Toolkit Channel Manager
Transceiver Toolkit User Interface

- **System Explorer**—displays design components and hardware connections.
- **Channel Manager**—control multiple channels simultaneously.
- **Tcl Console**—script control of the Transceiver Toolkit.
- **Messages**—displays information, warning, and error messages.

Quick Start

The Transceiver Toolkit user interface helps you to visualize and debug transceiver links in your design. To launch the toolkit, click **Tools > System Debugging Tools > Transceiver Toolkit**. Alternatively, you can run Tcl scripts from the command-line:

```
system-console --script=<name of script>
```

Get started quickly by downloading Transceiver Toolkit design examples from the [On-Chip Debugging Design Examples](#) website. For an online demonstration of how to use the Transceiver Toolkit to run a high-speed link test with one of the design examples, refer to the [Transceiver Toolkit Online Demo](#) on the Altera website.

Transceiver Debugging Flow

Testing transceiver links involves configuring your system for debug, and then running various link tests.

**Table 7-1: Transceiver Link Debugging Flow**

<table>
<thead>
<tr>
<th>System Configuration Steps</th>
<th>Flow Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Use one of the following methods to define a system that includes necessary transceiver debugging components:</td>
<td>1. Use one of the following methods to define a system that includes necessary transceiver debugging components:</td>
</tr>
<tr>
<td>• Download Transceiver Toolkit design examples from the <a href="#">On-Chip Debugging Design Examples</a> website. Modify Altera design examples to fit your design.</td>
<td>• Download Transceiver Toolkit design examples from the <a href="#">On-Chip Debugging Design Examples</a> website. Modify Altera design examples to fit your design.</td>
</tr>
<tr>
<td>• Integrate debugging components into your own design. Click <strong>Tools &gt; IP Catalog</strong> to select IP cores and define them in the parameter editor.</td>
<td>• Integrate debugging components into your own design. Click <strong>Tools &gt; IP Catalog</strong> to select IP cores and define them in the parameter editor.</td>
</tr>
<tr>
<td>2. Click <strong>Assignments &gt; Pin Planner</strong> to assign device I/O pins to match your device and board.</td>
<td>2. Click <strong>Assignments &gt; Pin Planner</strong> to assign device I/O pins to match your device and board.</td>
</tr>
<tr>
<td>3. Click <strong>Tools &gt; BluePrint Platform Designer</strong> &gt; to plan a legal device periphery floorplan.</td>
<td>3. Click <strong>Tools &gt; BluePrint Platform Designer</strong> &gt; to plan a legal device periphery floorplan.</td>
</tr>
<tr>
<td>4. Click <strong>Processing &gt; Start Compilation</strong> to compile your design.</td>
<td>4. Click <strong>Processing &gt; Start Compilation</strong> to compile your design.</td>
</tr>
<tr>
<td>5. Connect your target device to Altera programming hardware.</td>
<td>5. Connect your target device to Altera programming hardware.</td>
</tr>
<tr>
<td>6. Click <strong>Tools &gt; Programmer</strong> and then program the target device with the debugging system.</td>
<td>6. Click <strong>Tools &gt; Programmer</strong> and then program the target device with the debugging system.</td>
</tr>
</tbody>
</table>
### Flow Description

<table>
<thead>
<tr>
<th>Link Debugging Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Click <strong>File &gt; Load Design</strong>, and select the SRAM Object File (.sof) generated for your transceiver design. If you start the toolkit while a project is open, the project loads in the toolkit automatically.</td>
</tr>
<tr>
<td>2. (Optional) Create additional links between transmitter and receiver channels.</td>
</tr>
<tr>
<td>3. Run any of the following tests:</td>
</tr>
<tr>
<td>• Run BER with various combinations of PMA settings.</td>
</tr>
<tr>
<td>• Run PRBS Signal eye tests.</td>
</tr>
<tr>
<td>• Run custom traffic tests.</td>
</tr>
<tr>
<td>• Run link optimization tests.</td>
</tr>
<tr>
<td>• Directly control PMA analog settings to experiment with settings while the link is running.</td>
</tr>
</tbody>
</table>

### Related Information

**BluePrint Platform Planning**

### Configuring Systems for Transceiver Debug

To debug transceivers, you must first configure a system that includes the appropriate Altera IP core(s) that support each debugging operation. You can either modify an Altera design example, or configure your own system with required debugging IP components. The debugging system configuration varies by device family. Refer to the appropriate setup information to correctly configure or adapt a system for your target device.

### Configuring an Altera Design Example

Altera provides design examples to help you quickly test your own design. You can experiment with Altera design examples and modify them for your own application. Refer to the `readme.txt` of each design example for more information. Download the Transceiver Toolkit design examples from the On-Chip Debugging Design Examples page of the Altera website.

Use the design examples as a starting point to work with a particular signal integrity development board. The design examples provide the components to quickly test the functionality of the receiver and transmitter channels in your design. Change the transceiver settings in the design examples and observe the effects on transceiver link performance. Isolate and verify the high-speed serial links without debugging other logic in your design. You can modify and customize the design examples to match your intended transceiver design.

Once you download the design examples, open the Quartus Prime and click **Project > Restore Archived Project** to restore the design example project archive. If you have access to the same development board with the same device as mentioned in the `readme.txt` file of the example, you can directly program the device with the provided programming file in that example. If you want to recompile the design, you must make your modifications to the system configuration in Qsys, regenerate in Qsys, and recompile the design in the Quartus Prime software to generate a new programming file.
If you have the same board as mentioned in the readme.txt file, but a different device on your board, you must choose the appropriate device and recompile the design. For example, some early development boards are shipped with engineering sample devices.

You can make changes to the design examples so that you can use a different development board or a different device. If you have a different board, you must edit the necessary pin assignments and recompile the design examples.

Related Information
- On-Chip Debugging Design Examples

Configuring Your Own Debugging System
Rather than modifying the Altera Debugging Design Examples, you can integrate debugging IP components into your own design. Refer to the appropriate system configuration steps for your target device.

Related Information
Arria 10 Debug System Configuration on page 7-4

Arria 10 Debug System Configuration
For Arria 10 designs, the Transceiver Toolkit configuration requires instantiation of the Arria 10 Transceiver Native PHY IP core, and use of the built-in Altera Debug Master Endpoint (ADME). Qsys system generation automatically instantiates the JTAG debug link. You enable the ADME features by enabling specific parameters in the Transceiver Native PHY IP core.

Click Tools > IP Catalog to parameterize, generate, and instantiate the following debugging components.

Table 7-2: Arria 10 / 20nm Transceiver Toolkit IP Core Configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>Debugging Functions</th>
<th>Parameterization Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transceiver Native PHY</td>
<td>Supports all debugging functions</td>
<td>On the Dynamic Reconfiguration tab:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable dynamic reconfiguration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable odi acceleration logic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable Altera Debug Master Endpoint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable capability registers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable control and status registers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable prbs soft accumulators (enables hard PRBS Generator and Checker).</td>
</tr>
<tr>
<td>Transceiver ATX PLL</td>
<td>Required for Arria 10</td>
<td>On the Dynamic Reconfiguration tab:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable dynamic reconfiguration (required for System Console read/write access).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Turn on Enable Altera Debug Master Endpoint (required for System Console read/write access).</td>
</tr>
<tr>
<td>Transceiver PHY Reset Controller</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Enabling Altera Master Debug Endpoint (Arria 10)

The Altera Debug Master Endpoint (ADME) connects to the host link via the system level debug fabric for debugging Arria 10 designs. The ADME provides Avalon-MM Master capability, and is discoverable by System Console. To enable ADME for Arria 10 designs, you must set specific parameters when defining the Arria 10 Transceiver Native PHY and Arria 10 Transceiver ATX PLL IP cores. Click **Tools > IP Catalog** to select and define the following IP core parameters. Once properly configured, project synthesis inserts the ADME, debug fabric, and embedded logic.

<table>
<thead>
<tr>
<th>Component</th>
<th>Debugging Functions</th>
<th>Parameterization Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera Debug Master Endpoint (ADME)</td>
<td>• Supports control of PMA analog settings, ADCE settings, DFE settings, and EyeQ. &lt;br&gt;• discovers PHY and use toolkit on designs not using Qsys. &lt;br&gt;• Optionally, turn off to save resource count. (only an option if you instantiate the JTAG to Avalon Master Bridge. Otherwise, Transceiver Toolkit cannot function).</td>
<td>• Enabled from the Arria 10 Transceiver Native PHY Parameter Editor, <strong>Dynamic Reconfiguration</strong> tab.</td>
</tr>
<tr>
<td>JTAG Debug Link</td>
<td>Required for Arria 10.</td>
<td>For Qsys projects, the JTAG Debug Link is auto-instantiated.</td>
</tr>
</tbody>
</table>
Figure 7-2: Altera Debug Master Endpoint Block Diagram

Table 7-3: Enabling ADME IP Core Parameters for Arria 10 Designs

<table>
<thead>
<tr>
<th>IP Core</th>
<th>Parameter Setting</th>
<th>Location</th>
</tr>
</thead>
</table>
| Arria 10 Transceiver Native PHY | • Turn on Enable dynamic reconfiguration  
                              | • Turn on Enable Altera Debug Master Endpoint  
                              | • Turn on Enable prbs soft accumulators          | Dynamic Reconfiguration tab |
| Arria 10 Transceiver ATX PLL   | • Turn on Enable capability registers  
                              | • Turn on Enable control and status registers     | Dynamic Reconfiguration tab |
Use the following system configuration for BER, PRBS, and link optimization testing in Arria 10 devices. To perform BER or PRBS testing of Arria 10 designs, use the PRBS Generator and PRBS Checker functions of the Transceiver Native PHY IP core. You enable the built-in hard PRBS Generator and Checker by turning on Enable prbs soft accumulators when configuring the IP core. The Transceiver Toolkit performs required read-write-modify operations on the tested channel(s).
**Note:** For more information, refer to the *Arria 10 Transceiver Native PHY User Guide, Enabling the PRBS and Square Wave Data Generator and Enabling the PRBS Data Checker sections in the Reconfiguration Interface and Dynamic Configuration chapter.*

The built-in hard PRBS Data Pattern Generator and Checker requires no separate IP instantiation. The hard PRBS Data Pattern Generator and Checker does not support error injection. To use error injection for Arria 10 designs, use the Data Pattern Generator and Checker IP cores in your Arria 10 system design configuration.

**Figure 7-5: BER, PRBS, and Link Optimization Test Configuration (Arria 10)**

**Table 7-4: System Connections: BER and PRBS Testing (Arria 10)**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your design logic</td>
<td>• Transceiver Native PHY Hard PRBS Generator</td>
</tr>
<tr>
<td></td>
<td>• Transceiver Native PHY Hard PRBS Checker</td>
</tr>
</tbody>
</table>
Custom Traffic Signal Eye Test Configuration (Arria 10)
Use the following configuration for custom traffic signal eye tests in Arria 10 devices.

Figure 7-6: Custom Traffic Signal Eye Test Configuration for Arria 10 / Generation 10 / 20nm

Table 7-5: System Connections: Custom Traffic Signal Eye Test (Arria 10)

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your design logic custom traffic</td>
<td>Transceiver Native PHY</td>
</tr>
</tbody>
</table>

PMA Analog Setting Control Configuration (Arria 10)
Use the following configuration to control PMA analog settings in Arria 10 devices.

Figure 7-7: System Configuration: PMA Analog Setting Control (Arria 10)
Managing Transceiver Channels

The Channel Manager allows you to configure and control large numbers of channels in a spreadsheet view. You can view all the PMA and sweep settings for all channels. You can copy, paste, import, and export settings to and from channels. You can also start and stop sweeps for any or all channels. Right-click in the Channel Manager to view additional channel commands. The columns in the Channel Manager are movable, resizable, and sortable.

Figure 7-8: Channel Manager GUI

Copying and Pasting Settings

You can copy PMA and/or sweep settings from a selected row. You can paste PMA and/or sweep settings to one or more rows.

Importing and Exporting Settings

You can select a row in the Channel Manager to export your PMA settings to a text file. You can then select one or more rows in the Channel Manager to apply the PMA settings from a text file. The PMA settings in the text file apply to a single channel. When you import the PMA settings from a text file, you are duplicating one set of PMA settings for all selected channels.

Starting and Stopping Tests

The Channel Manager allows you to start and stop tests by right-clicking the channels. You can select several rows in the Channel Manager to start or stop test for multiple channels.
**Channel Display Modes**

The three display modes are **Current**, **Min/Max**, and **Best**. The default display mode is **Current**.

- **Current**—shows the current values from the device. The blue color text indicates that the settings are live.
- **Min/Max**—shows the minimum and maximum values to be used in the auto sweep.
- **Best**—shows the best tested values from the last completed auto sweep run.

**Note:** The **Transmitter Channels** tab only shows the **Current** display mode. Auto sweep cannot be performed on only a transmitter channel; a receiver channel is required to perform an auto sweep test.

**Creating Links**

The toolkit automatically creates links when a receiver and transmitter share a transceiver channel. You can also manually create and delete links between transmitter and receiver channels. You create links in the **Setup** dialog.

**Setup Dialog**

Click **Setup** from the **Channel Manager** to open the **Setup** dialog box.

**Table 7-6: Setup Dialog Popup Menu**

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Action When Clicked</th>
<th>Enabled If</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Transmitter Alias</td>
<td>Starts the inline edit of the alias of the selected row.</td>
<td>Only enabled if one row is selected.</td>
</tr>
<tr>
<td>Edit Receiver Alias</td>
<td>Starts the inline edit of the alias of the selected row.</td>
<td>Only enabled if one row is selected.</td>
</tr>
<tr>
<td>Edit Transceiver Link Alias</td>
<td>Starts the inline edit of the alias of the selected row.</td>
<td>Only enabled if one row is selected.</td>
</tr>
<tr>
<td>Copy</td>
<td>Copies the text of the selected row(s) to the clipboard. The text copied depends on the column clicked on. The text copied to the clipboard is newline delimited.</td>
<td>Enabled if one or more rows are selected.</td>
</tr>
</tbody>
</table>

**Controlling Transceiver Channels**

You can directly control and monitor transmitters, receivers, and links running on the board in real time. You can transmit a data pattern across the transceiver link, and then report the signal quality of the received data in terms of bit error rate or eye margin with EyeQ.

Click **Control Transmitter Channel** *(Transmitter Channels tab)*, **Control Receiver Channel** *(Receiver Channels tab)*, or **Control Transceiver Link** *(Transceiver Links tab)* to adjust transmitter or receiver settings while the channels are running.

**Debugging Transceiver Links**

The Transceiver Toolkit allows you to control and monitor the performance of high-speed serial links running on your board in real-time. You can identify the transceiver links in your design, transmit a data
pattern across the transceiver link, and report the signal quality of the received data in terms of bit error rate, bathtub curve, heat map, or EyeQ graph (for supported families).

The toolkit automatically identifies the transceiver links in your design, or you can manually create transceiver links. You can then run auto sweep to help you quickly identify the best PMA settings for each link. You can directly control the transmitter/receiver channels to experiment with various settings suggested by auto sweep. The EyeQ graph allows you to visualize the estimated horizontal and vertical eye opening at the receiver.

The Transceiver Toolkit supports various transceiver link testing configurations. You can identify and test the transceiver link between two Altera devices, or you can transmit a test pattern with a third-party device and monitor the data on an Altera device receiver channel. If a third-party chip includes self-test capability, then you can send the test pattern from the Altera device and monitor the signal integrity at the third-party device receiver channel. If the third-party device supports reverse serial loopback, you can run the test entirely within the Transceiver Toolkit.

Before you can monitor transceiver channels, you must configure a system with debugging components, and program the design into an FPGA. Once those steps are complete, use the following flow to test the channels:

1. Load the design in Transceiver Toolkit
2. Link hardware resources
3. Verify hardware connections
4. Identify transceiver channels
5. Run link tests or control PMA analog settings
6. View results

Step 1: Load Your Design

The Transceiver Toolkit automatically loads the last compiled design upon opening. To load any design into the toolkit, click File > Load Design and select the .sof programming file generated for your transceiver design. Loading the .sof automatically links the design to the target hardware in the toolkit. The System Explorer displays information about the loaded design.

Step 2: Link Hardware Resources

The toolkit automatically discovers connected hardware and designs. You can also manually link a design to connected hardware resources in the System Explorer.

If you are using more than one Altera board, you can set up a test with multiple devices linked to the same design. This setup is useful when you want to perform a link test between a transmitter and receiver on two separate devices. You can also load multiple Quartus Prime projects and make links between different systems. You can perform tests on completely separate and unrelated systems in a single tool instance.

Note: Prior to the Transceiver Toolkit version 11.1, you must manually load and link your design to hardware. In version 11.1 and later, the Transceiver Toolkit automatically links any device programmed with a project.
To link one design to one device by one USB-Blaster download cable, follow these steps:

1. Load the design for your Quartus Prime project.
2. Link each device to an appropriate design if the design has not auto-linked.
3. Create the link between channels on the device to test.
Linking Two Designs to Two Devices

To link two designs to two separate devices on the same board, connected by one USB-Blaster download cable, follow these steps:

1. Load the design for all the Quartus Prime project files you might need.
2. Link each device to an appropriate design if the design has not auto-linked.
3. Open the project for the second device.
4. Link the second device on the JTAG chain to the second design (unless the design auto-links).
5. Create a link between the channels on the devices you want to test.

Linking Designs and Devices on Separate Boards

To link two designs to two separate devices on separate boards, connected to separate USB-Blaster download cables, follow these steps:

1. Load the design for all the Quartus Prime project files you might need.
2. Link each device to an appropriate design if the design has not auto-linked.
3. Create the link between channels on the device to test.
4. Link the device you connected to the second USB-Blaster download cable to the second design.
5. Create a link between the channels on the devices you want to test.

Linking One Design on Two Devices

To link the same design on two separate devices, follow these steps:

1. In the Transceiver Toolkit, open the .sof you are using on both devices.
2. Link the first device to this design instance.
3. Link the second device to the design.
4. Create a link between the channels on the devices you want to test.

Step 3: Verify Hardware Connections

After you load your design and link your hardware, verify that the channels are connected correctly and looped back properly on the hardware. Use the toolkit to send data patterns and receive them correctly. Verifying your link and correct channel before you perform Auto Sweep or EyeQ tests can save time in the workflow.

After you have verified that the transmitter and receiver are communicating with each other, you can create a link between the two transceivers so that you can perform Auto Sweep and EyeQ tests with this pair.

Step 4: Identify Transceiver Channels

The Transceiver Toolkit automatically displays recognized transmitter and receiver channels. The toolkit identifies a channel automatically whenever a receiver and transmitter share a transceiver channel. You can also manually identify the transmitter and receiver in a transceiver channel and create a link between the two for testing.
When you run link tests, channel color highlights indicate the test status:

**Table 7-7: Channel Color Highlights**

<table>
<thead>
<tr>
<th>Color</th>
<th>Transmitter Channel</th>
<th>Receiver Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Channel is closed or generator clock is not running</td>
<td>Channel is closed or checker clock is not running</td>
</tr>
<tr>
<td>Green</td>
<td>Generator is sending a pattern</td>
<td>Checker is checking and data pattern is locked</td>
</tr>
<tr>
<td>Neutral</td>
<td>Channel is open, generator clock is running, and generator is not sending a pattern</td>
<td>Channel is open, checker clock is running, and checker is not checking</td>
</tr>
<tr>
<td>Yellow</td>
<td>N/A</td>
<td>Checker is checking and data pattern is not locked</td>
</tr>
</tbody>
</table>

**Step 5: Run Link Tests**

Once you identify the transceiver channels for debugging, you can run various link tests in the toolkit.

Use the **Transceiver Links** tab to control link tests. For example, use the Auto Sweep feature to sweep transceiver settings to determine the parameters that support the best BER value. Click **Link Auto Sweep**, **Link EyeQ** or **Link Auto Sweep & EyeQ** to adjust the PMA settings and run tests.

**Running BER Tests**

You can run BER tests across your transceiver link. After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run BER tests:

1. Click **Setup**.
   a. Select the generator and checker you want to control.
   b. Select the transmitter and receiver pair you want to control.
   c. Click **Create Transceiver Link** and click **Close**
2. Click **Control Transceiver Link**, and specify a PRBS Test pattern and Data pattern checker for **Checker mode**. The checker mode option is only available after you turn on **Enable EyeQ block** and **Enable Bit Error Rate Block** in the Reconfiguration Controller component.
   If you select **Bypass** for the Test pattern, the toolkit bypasses the PRBS generator and runs your design through the link. The bypass option is only available after you turn on **Enable Bypass interface** in the Reconfiguration Controller component.
3. Experiment with **Reconfiguration**, **Generator**, or **Checker** settings.
4. Click **Start** to run the pattern with your settings. You can then click **Inject Error** to inject error bits, **Reset** the counter, or **Stop** the test.
   **Note:** Arria 10 devices do not support **Inject Error** if you use the hard PRBS Pattern Generator and Checker in the system configuration.

**Related Information**

**Link Testing Configuration (Arria 10)** on page 7-7
Signal Eye Margin Testing

Some Altera devices include EyeQ circuitry that allows visualization of the horizontal and vertical eye margin at the receiver. For supported devices, use signal eye tests to tune the PMA settings of your transceiver. This results in the best eye margin and BER at high data rates. The toolkit disables signal eye testing for unsupported devices.

The EyeQ graph can display a bathtub curve, eye diagram representing eye margin, or heat map display. The run list displays the statistics of each EyeQ test. When PMA settings are suitable, the bathtub curve is wide, with sharp slopes near the edges. The curve is up to 30 units wide. If the bathtub is narrow, then the signal quality is poor. The wider the bathtub curve, the wider the eye. The smaller the bathtub curve, the smaller the eye. The eye contour shows the estimated horizontal and vertical eye opening at the receiver.

You can right-click any of the test runs in the list, and then click **Apply Settings to Device** to quickly apply those PMA settings to your device. You can also click **Export**, **Import**, or **Create Report**.

**Figure 7-11: EyeQ Settings and Status Showing Results of Two Test Runs**
Running PRBS Signal Eye Tests

You can run PRBS signal eye tests to visualize the estimated horizontal and vertical eye opening at the receiver. After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run PRBS signal eye tests:

1. Click Setup.
   a. Select the generator and checker you want to control.
   b. Select the transmitter and receiver pair you want to control.
   c. Click Create Transceiver Link and click Close.
2. Click Link EyeQ, and select EyeQ as the Test mode. The EyeQ mode displays test results as a bathtub curve, heat map, or eye contour representing bit error and phase offset data.
3. Specify Run length and EyeQ settings to control the test coverage and type of EyeQ results displayed, respectively.
4. Click Start to run the pattern with your settings. EyeQ uses the current channel settings to start a phase sweep of the channel. The phase sweep runs 32 iterations. As the run progresses, view the status under EyeQ status. Use this diagram to compare PMA settings for the same channel and to choose the best combination of PMA settings for a particular channel.
5. When the run completes, the chart displays the characteristics of each run. Click Stop to halt the test, change the PMA settings, and re-start the test. Click Create Report to export data to a table format for further viewing.

Related Information
Link Testing Configuration (Arria 10) on page 7-7

Running Custom Traffic Tests

After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run custom traffic tests:
1. Click Setup and specify the following:
   a. Select the associated reconfiguration controller.
   b. Click Create Transceiver Link and click Close.

2. Click the Receiver EyeQ, and select EyeQ as the Test mode. The EyeQ mode displays test results as a bathtub curve, heat map, or eye contour representing bit error and phase offset data.

3. Specify the PRBS Test pattern.

4. For Checker mode, select Serial bit comparator.
   The checker mode option is only available after you turn on Enable EyeQ block and Enable Bit Error Rate Block for the Reconfiguration Controller component.

5. Specify Run length and EyeQ settings to control the test coverage and type of EyeQ results displayed, respectively.

6. Click Start to run the pattern with your settings. EyeQ uses the current channel settings to start a phase sweep of the channel. The phase sweep runs 32 iterations. As the run progresses, view the status under EyeQ status.

7. When the run completes, the chart displays the characteristics of each run. Click Stop to halt the test, change the PMA settings, and re-start the test. Click Create Report to export data to a table format for further viewing.

Related Information
Custom Traffic Signal Eye Test Configuration (Arria 10) on page 7-9

Auto Sweep Testing
Use the auto sweep feature to automatically sweep ranges for the best transceiver PMA settings. Store a history of the test runs and keep a record of the best PMA settings. Use the best settings the toolkit determines in your final design for improved signal integrity.

Running Link Optimization Tests
After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run link optimization tests:

1. Click the Transceiver Links tab, and select the channel you want to control.
2. Click Link Auto Sweep. The Advanced tab appears with Auto sweep as Test mode.
3. Specify the PRBS Test pattern.
4. Specify Run length, experiment with the Transmitter settings, and Receiver settings to control the test coverage and PMA settings, respectively.
5. Click Start to run all combinations of tests meeting the PMA parameter limits.
6. When the run completes the chart is displayed and the characteristics of each run are listed in the run list. You can click Stop to halt the test, change the PMA settings, and re-start the test. Click Create Report to export data to a table format for further viewing.
7. To use decision feedback equalization (DFE) to determine the best tap settings, follow these steps:
   a. Use Auto Sweep to find optimal PMA settings while leaving the DFE mode set to Off.
   b. If BER = 0, use the best PMA settings achieved.
   c. If BER > 0, use this PMA setting, and set the minimum and maximum values obtained from Auto Sweep to match this setting. Set the maximum DFE range to limits for each of the three DFE settings.
   d. Run Create Report to view the results and determine which DFE setting has the best BER. Use these settings in conjunction with the PMA settings for the best results.
Controlling PMA Analog Settings

You can directly control PMA analog settings to experiment with settings while the link is running. To control PMA analog settings, follow these steps:

1. Click Setup.
   a. Click the Transmitter Channels tab, define a transmitter without a generator, and click Create Transmitter Channel.
   b. Click the Receiver Channels tab, define a receiver without a generator, and click Create Receiver Channel.
   c. Click the Transceiver Links tab, select the transmitter and receivers you want to control, and click Create Transceiver Link.
   d. Click Close.

2. Click Control Receiver Channel, Control Transmitter Channel, or Control Transceiver Link to directly control the PMA settings while running.
Figure 7-13: Controlling Transmitter Channel
**Figure 7-14: Controlling Receiver Channel**

![Control Panel](image.png)

**Basic**

- **Test pattern**: NRZ
- **Checker mode**: Hard NRZ
- **Loopback mode**: Off
- **Receiver channel**: RX \_dual \_B0000 \_address: \_11

**Transceiver**

- **Channel address**: 11
- **Data rate**: 10.250.00 Mbps
- **PLL refclk frequency**: 0.00 MHz
- **Enable word aligner**: Off
- **RX CDR locked to ref clock**: N/A
- **RX CDR locked to data**: N/A

**Reconfiguration**

- **Channel address**: 11
- **DC gain**: 2
- **Equalization mode**: Manual
- **Equalization control**: 21
- **VGA**: 2
- **DFE mode**: Manual
- **DFE 1st post-tap**: 6
- **DFE 2nd post-tap**: 5
- **DFE 3rd post-tap**: 4
- **DFE 4th post-tap**: 3
- **DFE 5th post-tap**: 2
- **DFE 6th post-tap**: 1
- **DFE 7th post-tap**: 0
- **DFE 8th post-tap**: 0
- **DFE 9th post-tap**: 0
- **DFE 10th post-tap**: 0
- **DFE 11th post-tap**: 0

**Checker**

- **Number of bits tested**: 5.512E11
- **Number of error bits**: 8
- **Bit error rate (BER)**: 1.4512E-11

---

*Debugging Transceiver Links*  
*Altera Corporation*  
*Send Feedback*
Related Information

PMA Analog Setting Control Configuration (Arria 10) on page 7-9
Troubleshooting Common Errors

- **Missing high-speed link pin connections**
  
  The pin connections to identify high-speed links (tx_p/n and rx_p/n) could be missing. When porting an older design to the latest version of the Quartus Prime software, please make sure your connections were preserved.

- **Reset Issues**
  
  Ensure that the reset input to the Transceiver Native PHY, Transceiver Reset Controller, and ATX PLL IP cores is not held active (1'b1). The Transceiver Toolkit highlights in red, all the Transceiver Native PHY channels which you are trying to set up.

- **Unconnected reconfig_clk**
  
  The reconfig_clk input to the Transceiver Native PHY and ATX PLL IP cores need to be connected and driven. Otherwise, the transceiver link channel will not be displayed.

User Interface Settings Reference

The following settings are available for interaction with the transmitter channels or receiver channels or transceiver links in the Transceiver Toolkit user interface.

**Table 7-8: Transceiver Toolkit Control Panel Settings**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Control Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias</td>
<td>Name you choose for the channel.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Auto sweep status</td>
<td>Reports the current and best tested bits, errors, bit error rate, and case count for the current auto sweep test.</td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Bit error rate (BER)</td>
<td>Specifies errors divided by bits tested since the last reset of the checker.</td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Channel address</td>
<td>Logical address number of the transceiver channel.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Control Panel</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
</tbody>
</table>
| Checker mode | Specify **Data pattern checker** or **Serial bit comparator** for BER tests. If you enable **Serial bit comparator** the Data Pattern Generator sends the PRBS pattern, but the pattern is checked by the serial bit comparator. In **Bypass mode**, clicking **Start** begins counting on the Serial bit comparator. For BER testing:  
- Arria 10 devices supports the Data Pattern Checker and the Hard PRBS. For EyeQ testing:  
- Arria 10 devices Arria 10 support the Serial Bit Checker.  
- Stratix V devices support the Data Pattern Checker and the Serial Bit Checker. | Receiver  
Transceiver Link                                    |
| Data rate    | Data rate of the channel as read from the project file or data rate as measured by the frequency detector. To use the frequency detector, turn on **Enable Frequency Counter** in the Data Pattern Checker IP core and/or Data Pattern Generator IP core, regenerate the IP cores, and recompile the design. The measured data rate depends on the Avalon management clock frequency as read from the project file. Click the refresh button next to the measured **Data rate** if you make changes to your settings and want to sample the data rate again. | Transmitter  
Receiver  
Transceiver Link                                      |
| DC gain      | Circuitry that provides an equal boost to the incoming signal across the frequency spectrum.                                                                                                                                                  | Receiver  
Transceiver Link                                      |
| DFE mode     | Decision feedback equalization (DFE) for improving signal quality. DFE modes are **Off**, **Manual** and **Continuous Adaptation** in Arria 10 devices. Continuous mode DFE automatically tries to find the best tap values.                                                                 | Receiver  
Transceiver Link                                      |
<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Control Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equalization control</td>
<td>Boosts the high-frequency gain of the incoming signal, thereby compensating for the low-pass filter effects of the physical medium. When used with DFE, use DFE in <strong>Manual</strong> or <strong>Continuous</strong> mode.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Equalization mode</td>
<td>You can set Equalization Mode to <strong>Manual</strong> or <strong>Triggered</strong> for Arria 10 devices.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Error rate limit</td>
<td>Turns on or off error rate limits. <strong>Start checking after</strong> waits until the set number of bits are satisfied until it starts looking at the bit error rate (BER) for the next two checks. <strong>Bit error rate achieves below</strong> sets upper bit error rate limits. If the error rate is better than the set error rate, the test ends. <strong>Bit error rate exceeds</strong> Sets lower bit error rate limits. If the error rate is worse than the set error rate, the test ends.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>EyeQ mode</td>
<td>Allows you to specify Eye contour or Bathtub curve as the type of EyeQ graph generated by the test.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>EyeQ phase step</td>
<td>Sets the phase step for sampling the data from an offset of the CDR (clock data recovery) data path; set to Off to use the regular clock data recovery (CDR) data path.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>EyeQ status</td>
<td>Displays a graphical representation of signal integrity as an eye contour, bathtub curve plot, or heat map.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>EyeQ vertical step</td>
<td>Sets the voltage threshold of the sampler to report the height of the eye. Negative numbers are allowed for vertical steps to capture asymmetric eye.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Horizontal phase step</td>
<td>Specify the number of horizontal steps to increment when performing a sweep. Increasing the value increases the speed of the test but at a lower resolution. This option only applies to eye contour.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Control Panel</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Increase test range</td>
<td>Right-click in the Advanced panel to use the span capabilities of Auto Sweep to automatically increase the span of tests by one unit down for the minimum and one unit up for the maximum, for the selected set of controls. You can span either PMA Analog controls (non-DFE controls), or the DFE controls. You can quickly set up a test to check if any PMA setting combinations near your current best could yield better results.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Maximum tested bits</td>
<td>Sets the maximum number of tested bits for each test iteration.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Number of bits tested</td>
<td>Specifies the number of bits tested since the last reset of the checker.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Number of error bits</td>
<td>Specifies the number of error bits encountered since the last reset of the checker.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>PLL refclk freq</td>
<td>Channel reference clock frequency as read from the project file or measured reference clock frequency as calculated from the measured data rate.</td>
<td>Transmitter&lt;br&gt;Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Populate with</td>
<td>Right-click in the Advanced panel to load current values on the device as a starting point, or initially load the best settings determined through auto sweep. The Quartus Prime software automatically applies the values you specify in the drop-down lists for the Transmitter settings and Receiver settings.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Preamble word</td>
<td>Word to send out if you use the preamble mode (only if you use soft PRBS Data Pattern Generator and Checker).</td>
<td>Transmitter&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Pre-emphasis</td>
<td>The programmable pre-emphasis module in each transmit buffer boosts high frequencies in the transmit data signal, which may be attenuated in the transmission media. Using pre-emphasis can maximize the data eye opening at the far-end receiver.</td>
<td>Transmitter&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Receiver channel</td>
<td>Specifies the name of the selected receiver channel.</td>
<td>Receiver&lt;br&gt;Transceiver Link</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Control Panel</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Refresh Button</td>
<td>After loading the .pof, loads fresh settings from the registers after running dynamic reconfiguration.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the current test.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Rules Based Configuration (RBC) validity checking</td>
<td>Displays invalid combination of settings in red in each list under <strong>Transmitter settings</strong> and <strong>Receiver settings</strong>, based on previous settings. If selected, the settings remain in red to indicate the currently selected combination is invalid. This avoids manually testing invalid settings that you cannot compile for your design. This prevents setting the device into an invalid mode for extended periods of time and potentially damaging the circuits.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Run length</td>
<td>Sets coverage parameters for test runs.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Run results table</td>
<td>Lists the statistics of each EyeQ test run.</td>
<td>Transmitter, Receiver, Transceiver Link</td>
</tr>
<tr>
<td>RX CDR PLL status</td>
<td>Shows the receiver in lock-to-reference (LTR) mode. When in auto-mode, if data cannot be locked, this signal alternates in LTD mode if the CDR is locked to data.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>RX CDR data status</td>
<td>Shows the receiver in lock-to-data (LTD) mode. When in auto-mode, if data cannot be locked, the signal stays high when locked to data and never toggles.</td>
<td>Receiver, Transceiver Link</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Control Panel</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Serial loopback enabled</td>
<td>Inserts a serial loopback before the buffers, allowing you to form a link on a transmitter and receiver pair on the same physical channel of the device.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Start</td>
<td>Starts the pattern generator or checker on the channel to verify incoming data.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops generating patterns and testing the channel.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Target bit error rate</td>
<td>Finds the contour edge of the bit error rate that you select. This option only applies to eye contour mode.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Test mode</td>
<td>Allows you to specify the <strong>Auto sweep</strong>, <strong>EyeQ</strong>, or <strong>Auto sweep and EyeQ</strong> test mode.</td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Test pattern</td>
<td>Test pattern sent by the transmitter channel. Arria 10 devices support <strong>PRBS9</strong>, <strong>PRBS15</strong>, <strong>PRBS23</strong>, and <strong>PRBS31</strong>.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Time limit</td>
<td>Specifies the time limit unit and value to have a maximum bounds time limit for each test iteration</td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Transmitter channel</td>
<td>Specifies the name of the selected transmitter channel.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>TX/CMU PLL status</td>
<td>Provides status of whether the transmitter channel PLL is locked to the reference clock.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Use preamble upon start</td>
<td>If turned on, sends the preamble word before the test pattern. If turned off, starts sending the test pattern immediately.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
<td>Control Panel</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>Vertical phase step</td>
<td>Specify the number of vertical steps to increment when performing a sweep. Increasing the value increases the speed of the test but at a lower resolution. This option only applies to the eye contour.</td>
<td>Transmitter</td>
</tr>
<tr>
<td>interval</td>
<td></td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
<tr>
<td><strong>VGA</strong></td>
<td>The variable gain amplifier (VGA) amplifies the signal amplitude and ensures a constant voltage swing before the data is fed to the CDR for sampling. This assignment controls the VGA output voltage swing and can be set to any value from 0 to 7.</td>
<td>Receiver</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Supports Arria 10 devices only</td>
<td>Transceiver Link</td>
</tr>
<tr>
<td><strong>V&lt;sub&gt;OD&lt;/sub&gt; control</strong></td>
<td>Programmable transmitter differential output voltage.</td>
<td>Transmitter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transceiver Link</td>
</tr>
</tbody>
</table>

### Scripting API Reference

You can alternatively use Tcl commands to access Transceiver Toolkit functions, rather than using the GUI. You can script various tasks, such as loading a project, creating design instances, linking device resources, and identifying high-speed serial links. You can save your project setup in a Tcl script for use in subsequent testing sessions. You can also build a custom test routine script.

After you set up and define links that describe the entire physical system, you can click **Save Tcl Script** to save the setup for future use. To run the scripts, double-click script names in the System Explorer scripts folder.

View a list of the available Tcl commands in the Tcl Console window. Select Tcl commands in the list to view descriptions, including example usage.

To view Tcl command descriptions from the Tcl Console window:

1. Type `help help`. The Console displays all Transceiver Toolkit Tcl commands.
2. Type `help <command name>`. The Console displays the command description.

### Transceiver Toolkit Commands

The following tables list the available Transceiver Toolkit scripting commands.

#### Table 7-9: Transceiver Toolkit Channel_rx Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_channel_rx_get_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of the current checker data. The results are in the order of number of bits, number of errors, and bit error rate.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_dcgain</td>
<td>&lt;service-path&gt;</td>
<td>Gets the DC gain value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_dfe_tap_value</td>
<td>&lt;service-path&gt; &lt;tap position&gt;</td>
<td>Gets the current tap value of the specified channel at the specified tap position.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_eqctrl</td>
<td>&lt;service-path&gt;</td>
<td>Gets the equalization control value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current data checker pattern by name.</td>
</tr>
<tr>
<td>transceiver_channel_rx_has_dfe</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether this channel has the DFE feature available.</td>
</tr>
<tr>
<td>transceiver_channel_rx_has_eyeq</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether the EyeQ feature is available for the specified channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_checking</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is running.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_dfe_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether the DFE feature is enabled on the specified channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_locked</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is locked onto the incoming data.</td>
</tr>
<tr>
<td>transceiver_channel_rx_reset_counters</td>
<td>&lt;service-path&gt;</td>
<td>Resets the bit and error counters inside the checker.</td>
</tr>
<tr>
<td>transceiver_channel_rx_reset</td>
<td>&lt;service-path&gt;</td>
<td>Resets the specified channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dcgain</td>
<td>&lt;service-path&gt; &lt;value&gt;</td>
<td>Sets the DC gain value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dfe_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables or disables the DFE feature on the specified channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dfe_tap_value</td>
<td>&lt;service-path&gt; &lt;tap position&gt; &lt;tap value&gt;</td>
<td>Sets the current tap value of the specified channel at the specified tap position to the specified value.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dfe_adaptive</td>
<td>&lt;service-path&gt;</td>
<td>Sets the mode of DFE adaptation. 0=off, 1=adaptive, 2= one-time adaptive</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eqctrl</code></td>
<td><code>&lt;service-path&gt; &lt;value&gt;</code></td>
<td>Sets the equalization control value on the receiver channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_start_checking</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Starts the checker.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_stop_checking</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Stops the checker.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_get_eyeq_phase_step</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the current phase step of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_pattern</code></td>
<td><code>&lt;service-path&gt; &lt;pattern-name&gt;</code></td>
<td>Sets the expected pattern to the one specified by the pattern name.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_eyeq_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets whether the EyeQ feature is enabled on the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eyeq_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables the EyeQ feature on the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eyeq_phase_step</code></td>
<td><code>&lt;service-path&gt; &lt;phase step&gt;</code></td>
<td>Sets the phase step of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables the word aligner of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Gets whether the word aligner feature is enabled on the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_locked</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns non-zero if the checker is locked onto the incoming signal.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_rx_locked_to_data</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns 1 if transceiver is in lock to data (LTD) mode. Otherwise 0.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_rx_locked_to_ref</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns 1 if transceiver is in lock to reference (LTR) mode. Otherwise 0.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_has_eyeq_1d</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Detects whether the eye viewer pointed to by <code>&lt;service-path&gt;</code> supports 1D-EyeQ mode.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_1deye_mode</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables 1D-EyeQ mode.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_get_1deye_mode</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the current on or off status of 1D-EyeQ mode.</td>
</tr>
</tbody>
</table>
Table 7-10: Transceiver Toolkit Channel _tx Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_channel_tx_disable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Disables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_enable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_number_of_preamble_beats</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the currently set number of beats to send out the preamble word.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_pattern</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the currently set pattern.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_preamble_word</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the currently set preamble word.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_preemph0t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis pre-tap value on the transmitter channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_preemph1t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis first post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_preemph2t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis second post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_vodctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the VOD control value on the transmitter channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_inject_error</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Injects a 1-bit error into the generator's output.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_is_generating</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns non-zero if the generator is running.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_is_preamble_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns non-zero if preamble mode is enabled.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_set_number_of_preamble_beats</code></td>
<td><code>&lt;service-path&gt;</code>&lt;number-of-preamble-beats&gt;</td>
<td>Sets the number of beats to send out the preamble word.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_pattern</td>
<td>&lt;service-path&gt;&lt;pattern-name&gt;</td>
<td>Sets the output pattern to the one specified by the pattern name.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preamble_word</td>
<td>&lt;service-path&gt;&lt;preamble-word&gt;</td>
<td>Sets the preamble word to be sent out.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph0t</td>
<td>&lt;service-path&gt;&lt;preemph0t value&gt;</td>
<td>Sets the pre-emphasis pre-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph1t</td>
<td>&lt;service-path&gt;&lt;preemph1t value&gt;</td>
<td>Sets the pre-emphasis first post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph2t</td>
<td>&lt;service-path&gt;&lt;preemph2t value&gt;</td>
<td>Sets the pre-emphasis second post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_vodctrl</td>
<td>&lt;service-path&gt;&lt;vodctrl value&gt;</td>
<td>Sets the V\textsubscript{OD} control value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_start_generation</td>
<td>&lt;service-path&gt;</td>
<td>Starts the generator.</td>
</tr>
<tr>
<td>transceiver_channel_tx_stop_generation</td>
<td>&lt;service-path&gt;</td>
<td>Stops the generator.</td>
</tr>
</tbody>
</table>

**Table 7-11: Transceiver Toolkit Transceiver Toolkit Debug_Link Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_debug_link_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Gets the currently set pattern the link uses to run the test.</td>
</tr>
<tr>
<td>transceiver_debug_link_is_running</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the test is running on the link.</td>
</tr>
<tr>
<td>transceiver_debug_link_set_pattern</td>
<td>&lt;service-path&gt; &lt;data pattern&gt;</td>
<td>Sets the pattern the link uses to run the test.</td>
</tr>
<tr>
<td>transceiver_debug_link_start_running</td>
<td>&lt;service-path&gt;</td>
<td>Starts running a test with the currently selected test pattern.</td>
</tr>
<tr>
<td>transceiver_debug_link_stop_running</td>
<td>&lt;service-path&gt;</td>
<td>Stops running the test.</td>
</tr>
</tbody>
</table>
**Table 7-12: Transceiver Toolkit Reconfig_Ananlog Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_reconfig_analog_get_logical_channel_address</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the transceiver logical channel address currently set.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_rx_dcgain</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the DC gain value on the receiver channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_rx_eqctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the equalization control value on the receiver channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph0t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph1t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph2t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_vodctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the V&lt;sub&gt;OD&lt;/sub&gt; control value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_setLogical_channel_address</code></td>
<td><code>&lt;service-path&gt;&lt;logical channel address&gt;</code></td>
<td>Sets the transceiver logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_set_rx_dcgain</code></td>
<td><code>&lt;service-path&gt;&lt;dc_gain value&gt;</code></td>
<td>Sets the DC gain value on the receiver channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_set_rx_eqctrl</code></td>
<td><code>&lt;service-path&gt;&lt;eqctrl value&gt;</code></td>
<td>Sets the equalization control value on the receiver channel specified by the current logical channel address.</td>
</tr>
</tbody>
</table>
### Table 7-13: Transceiver Toolkit Decision Feedback Equalization (DFE) Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph0t</td>
<td><code>&lt;service-path&gt;</code>&lt;preemph0t value&gt;</td>
<td>Sets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph1t</td>
<td><code>&lt;service-path&gt;</code>&lt;preemph1t value&gt;</td>
<td>Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph2t</td>
<td><code>&lt;service-path&gt;</code>&lt;preemph2t value&gt;</td>
<td>Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_vodctrl</td>
<td><code>&lt;service-path&gt;</code>&lt;vodctrl value&gt;</td>
<td>Sets the ( V_{OD} ) control value on the transmitter channel specified by the current logical channel address.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_xcvr_reconfig_dfe_get_logical_channel_address</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the logical channel address that other alt_xcvr_reconfig_dfe commands use to apply.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_is_enabled</td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets whether the DFE feature is enabled on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_enabled</td>
<td><code>&lt;service-path&gt;</code>&lt;disable(0)/enable(1)&gt;</td>
<td>Enables or disables the DFE feature on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_logical_channel_address</td>
<td><code>&lt;service-path&gt;</code>&lt;logical channel address&gt;</td>
<td>Sets the logical channel address that other alt_xcvr_reconfig_eye_viewer commands use.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_tap_value</td>
<td><code>&lt;service-path&gt;</code>&lt;tap position&gt;&lt;tap value&gt;</td>
<td>Sets the tap value at the previously specified channel at specified tap position and value.</td>
</tr>
</tbody>
</table>
### Table 7-14: Transceiver Toolkit Eye Monitor Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alt_xcvr_custom_is_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Gets whether the word aligner feature is enabled on the previously specified channel.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_set_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables the word aligner of the previously specified channel.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_is_rx_locked_to_data</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns whether the receiver CDR is locked to data.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_is_rx_locked_to_ref</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns whether the receiver CDR PLL is locked to the reference clock.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_is_serial_loopback_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns whether the serial loopback mode of the previously specified channel is enabled.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_set_serial_loopback_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables the serial loopback mode of the previously specified channel.</td>
</tr>
<tr>
<td><code>alt_xcvr_custom_is_tx_pll_locked</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns whether the transmitter PLL is locked to the reference clock.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_get_logical_channel_address</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the logical channel address on which other <code>alt_reconfig_eye_viewer</code> commands will use to apply.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_get_phase_step</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the current phase step of the previously specified channel.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_is_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets whether the EyeQ feature is enabled on the previously specified channel.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_set_enabled</code></td>
<td><code>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</code></td>
<td>Enables or disables the EyeQ feature on the previously specified channel. Setting a value of 2 enables both EyeQ and the Serial Bit Comparator.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_set_logical_channel_address</code></td>
<td><code>&lt;service-path&gt; &lt;logical channel address&gt;</code></td>
<td>Sets the logical channel address on which other <code>alt_reconfig_eye_viewer</code> commands will use to apply.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_set_phase_step</code></td>
<td><code>&lt;service-path&gt; &lt;phase step&gt;</code></td>
<td>Sets the phase step of the previously specified channel.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_has_ber_checker</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Detects whether the eye viewer pointed to by <code>&lt;service-path&gt;</code> supports the Serial Bit Comparator.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_is_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Detects whether the Serial Bit Comparator is enabled.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_start</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Starts the Serial Bit Comparator counters.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_stop</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Stops the Serial Bit Comparator counters.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_reset_counters</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Resets the Serial Bit Comparator counters.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_is_running</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets whether the Serial Bit Comparator counters are currently running or not.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_ber_checker_get_data</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the current total bit, error bit, and exception counts for the Serial Bit Comparator.</td>
</tr>
<tr>
<td><code>alt_xcvr_reconfig_eye_viewer_has_1deye</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Detects whether the eye viewer pointed to by <code>&lt;service-path&gt;</code> supports 1D-EyeQ mode.</td>
</tr>
</tbody>
</table>

*Debugging Transceiver Links*
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_set_1deye_mode</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables or disables 1D-EyeQ mode.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_get_1deye_mode</td>
<td>&lt;service-path&gt;</td>
<td>Gets the enable or disabled state of 1D-EyeQ mode.</td>
</tr>
</tbody>
</table>

Table 7-15: Channel Type Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_channel_type</td>
<td>&lt;service-path&gt;&lt;logical-channel-num&gt;</td>
<td>Reports the detected type (GX/GT) of channel &lt;logical-channel-num&gt; for the reconfiguration block located at &lt;service-path&gt;.</td>
</tr>
<tr>
<td>set_channel_type</td>
<td>&lt;service-path&gt;&lt;logical-channel-num&gt;&lt;channel-type&gt;</td>
<td>Overrides the detected channel type of channel &lt;logical-channel-num&gt; for the reconfiguration block located at &lt;service-path&gt; to the type specified (0:GX, 1:GT).</td>
</tr>
</tbody>
</table>

Table 7-16: Loopback Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
</table>
| loopback_get | <service-path> | Returns the value of a setting or result on the loopback channel. Available results include:  
- Status—running or stopped.  
- Bytes—number of bytes sent through the loopback channel.  
- Errors—number of errors reported by the loopback channel.  
- Seconds—number of seconds since the loopback channel was started. |
Data Pattern Generator Commands

You can use Data Pattern Generator commands to control data patterns for debugging transceiver channels. You must instantiate the Data Pattern Generator component to support these commands.

Table 7-17: Soft Data Pattern Generator Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_pattern_generator_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the data pattern generator.</td>
</tr>
<tr>
<td>data_pattern_generator_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the data pattern generator.</td>
</tr>
<tr>
<td>data_pattern_generator_is_generating</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the generator is running.</td>
</tr>
<tr>
<td>data_pattern_generator_inject_error</td>
<td>&lt;service-path&gt;</td>
<td>Injects a 1-bit error into the generator output.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>---------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>data_pattern_generator_set_pattern</code></td>
<td><code>&lt;service-path&gt; &lt;pattern-name&gt;</code></td>
<td>Sets the output pattern specified by the <code>&lt;pattern-name&gt;</code>. In all, 6 patterns are available, 4 are pseudo-random binary sequences (PRBS), 1 is high frequency and 1 is low frequency. The PRBS7, PRBS15, PRBS23, PRBS31, HF (outputs high frequency, constant pattern of alternating 0s and 1s), and LF (outputs low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols) pattern names are defined. PRBS files are clear text and you can modify the PRBS files.</td>
</tr>
<tr>
<td><code>data_pattern_generator_get_pattern</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns currently selected output pattern.</td>
</tr>
<tr>
<td><code>data_pattern_generator_get_available_patterns</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns a list of available data patterns by name.</td>
</tr>
<tr>
<td><code>data_pattern_generator_enable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>data_pattern_generator_disable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Disables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>data_pattern_generator_is_preamble_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns a non-zero value if preamble mode is enabled.</td>
</tr>
<tr>
<td><code>data_pattern_generator_set_preamble_word</code></td>
<td><code>&lt;preamble-word&gt;</code></td>
<td>Sets the preamble word (could be 32-bit or 40-bit).</td>
</tr>
<tr>
<td><code>data_pattern_generator_get_preamble_word</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the preamble word.</td>
</tr>
<tr>
<td><code>data_pattern_generator_set_preamble_beats</code></td>
<td><code>&lt;service-path&gt; &lt;number-of-preamble-beats&gt;</code></td>
<td>Sets the number of beats to send out in the preamble word.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>data_pattern_generator_get_preamble_beats</td>
<td>&lt;service-path&gt;</td>
<td>Returns the currently set number of beats to send out in the preamble word.</td>
</tr>
<tr>
<td>data_pattern_generator_fcnter_start</td>
<td>&lt;service-path&gt;&lt;max-cycles&gt;</td>
<td>Sets the max cycle count and starts the frequency counter.</td>
</tr>
<tr>
<td>data_pattern_generator_check_status</td>
<td>&lt;service-path&gt;</td>
<td>Queries the data pattern generator for current status. Returns a bitmap indicating the status, with bits defined as follows: [0]-enabled, [1]-bypass enabled, [2]-avalon, [3]-sink ready, [4]-source valid, and [5]-frequency counter enabled.</td>
</tr>
<tr>
<td>data_pattern_generator_fcnter_report</td>
<td>&lt;service-path&gt;&lt;force-stop&gt;</td>
<td>Reports the current measured clock ratio, stopping the counting first depending on &lt;force-stop&gt;.</td>
</tr>
</tbody>
</table>

**Table 7-18: Hard Data Pattern Generator Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard_prbs_generator_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the specified generator.</td>
</tr>
<tr>
<td>hard_prbs_generator_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the specified generator.</td>
</tr>
<tr>
<td>hard_prbs_generator_is_generating</td>
<td>&lt;service-path&gt;</td>
<td>Checks the generation status. Returns 1 if generating, 0 otherwise.</td>
</tr>
<tr>
<td>hard_prbs_generator_set_pattern</td>
<td>&lt;service-path&gt; &lt;pattern&gt;</td>
<td>Sets the pattern of the specified hard PRBS generator to parameter pattern.</td>
</tr>
<tr>
<td>hard_prbs_generator_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current pattern for a given hard PRBS generator.</td>
</tr>
<tr>
<td>hard_prbs_generator_get_available_patterns</td>
<td>&lt;service-path&gt;</td>
<td>Returns the available patterns for a given hard PRBS generator.</td>
</tr>
</tbody>
</table>
Data Pattern Checker Commands

You can use Data Pattern Checker commands to verify your generated data patterns. You must instantiate the Data Pattern Checker component to support these commands.

Table 7-19: Soft Data Pattern Checker Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_pattern_checker_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the data pattern checker.</td>
</tr>
<tr>
<td>data_pattern_checker_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the data pattern checker.</td>
</tr>
<tr>
<td>data_pattern_checker_is_checking</td>
<td>&lt;service-path&gt;</td>
<td>Returns a non-zero value if the checker is running.</td>
</tr>
<tr>
<td>data_pattern_checker_is_locked</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is locked onto the incoming data.</td>
</tr>
<tr>
<td>data_pattern_checker_set_pattern</td>
<td>&lt;service-path&gt; &lt;pattern-name&gt;</td>
<td>Sets the expected pattern to the one specified by the &lt;pattern-name&gt;.</td>
</tr>
<tr>
<td>data_pattern_checker_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Returns the currently selected expected pattern by name.</td>
</tr>
<tr>
<td>data_pattern_checker_get_available_patterns</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of available data patterns by name.</td>
</tr>
<tr>
<td>data_pattern_checker_get_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of the current checker data. The results are in the following order: number of bits, number of errors, and bit error rate.</td>
</tr>
<tr>
<td>data_pattern_checker_reset_counters</td>
<td>&lt;service-path&gt;</td>
<td>Resets the bit and error counters inside the checker.</td>
</tr>
<tr>
<td>data_pattern_checker_fcnter_start</td>
<td>&lt;service-path&gt; &lt;max-cycles&gt;</td>
<td>Sets the max cycle count and starts the frequency counter.</td>
</tr>
<tr>
<td>Command</td>
<td>Arguments</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>data_pattern_checker_fcnter_report</td>
<td>&lt;service-path&gt;&lt;force-stop&gt;</td>
<td>Reports the current measured clock ratio, stopping the counting first depending on &lt;force-stop&gt;.</td>
</tr>
</tbody>
</table>

**Table 7-20: Hard Data Pattern Checker Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard_prbs_checker_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the specified hard PRBS checker.</td>
</tr>
<tr>
<td>hard_prbs_checker_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the specified hard PRBS checker.</td>
</tr>
<tr>
<td>hard_prbs_checker_is_checking</td>
<td>&lt;service-path&gt;</td>
<td>Checks the running status of the specified hard PRBS checker. Returns a non-zero value if the checker is running.</td>
</tr>
<tr>
<td>hard_prbs_checker_set_pattern</td>
<td>&lt;service-path&gt; &lt;pattern&gt;</td>
<td>Sets the pattern of the specified hard PRBS checker to parameter &lt;pattern&gt;.</td>
</tr>
<tr>
<td>hard_prbs_checker_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current pattern for a given hard PRBS checker.</td>
</tr>
<tr>
<td>hard_prbs_checker_get_available_patterns</td>
<td>&lt;service-path&gt;</td>
<td>Returns the available patterns for a given hard PRBS checker.</td>
</tr>
<tr>
<td>hard_prbs_checker_get_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current bit and error count data from the specified hard PRBS checker.</td>
</tr>
<tr>
<td>hard_prbs_checker_reset_counters</td>
<td>&lt;service-path&gt;</td>
<td>Resets the bit and error counts of the specified hard PRBS checker.</td>
</tr>
</tbody>
</table>
### Document Revision History

#### Table 7-21: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added description of new Refresh button.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added description of VGA dialog box.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added two tables in Transceiver Toolkit Commands section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard Data Pattern Generator Commands</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard Data Pattern Checker Commands</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Separated Arria 10 and Stratix V system configuration steps.</td>
</tr>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Added description of new Refresh button.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added description of VGA dialog box.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added two tables in Transceiver Toolkit Commands section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard Data Pattern Generator Commands</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard Data Pattern Checker Commands</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Separated Arria 10 and Stratix V system configuration steps.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| May 2015   | 15.0.0  | • Added section about Implementation Differences Between Stratix V and Arria 10.  
• Added section about Recommended Flow for Arria 10 Transceiver Toolkit Design with the Quartus Prime Software.  
• Added section about Transceiver Toolkit Troubleshooting  
• Updated the following sections with information about using the Transceiver Toolkit with Arria 10 devices:  
  • Serial Bit Comparator Mode  
  • Arria 10 Support and Limitations  
  • Configuring BER Tests  
  • Configuring PRBS Signal Eye Tests  
  • Adapting Altera Design Examples  
  • Modifying Design Examples  
  • Configuring Custom Traffic Signal Eye Tests  
  • Configuring Link Optimization Tests  
  • Configuring PMA Analog Setting Control  
  • Running BER Tests  
  • Toolkit GUI Setting Reference  
• Reworked Table: Transceiver Toolkit IP Core Configuration  
• Replaced Figure: EyeQ Settings and Status Showing Results of Two Test Runs with Figure: EyeQ Settings and Status Showing Results of Three Test Runs.  
• Added Figure: Arria 10 Altera Debug Master Endpoint Block Diagram.  
• Added Figure: BER Test Configuration (Arria10/ Gen 10/ 20nm) Block Diagram.  
• Added Figure: PRBS Signal Test Configuration (Arria 10/ 20nm) Block Diagram.  
• Added Figure: Custom Traffic Signal Eye Test Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram.  
• Added Figure: PMA Analog Setting Control Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram.  
• Added Figure: One Channel Loopback Mode (Arria 10/ 20nm) Block Diagram.  
• Added Figure: Four Channel Loopback Mode (Arria 10/ Gen 10/ 20nm) Block Diagram.  

Software Version 15.0 Limitations

• Transceiver Toolkit supports EyeQ for Arria 10 designs.  
• Supports optional hard acceleration for EyeQ. This allows for much faster EyeQ data collection. Enable this in the Arria 10 Transceiver Native PHY IP core under the **Dynamic Configuration** tab. Turn on **Enable ODI acceleration logic**.  

---

Debugging Transceiver Links

Alterna Corporation

Send Feedback
## Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December, 2014</td>
<td>14.1.0</td>
<td>• Added section about Arria 10 support and limitations.</td>
</tr>
<tr>
<td>June, 2014</td>
<td>14.0.0</td>
<td>• Updated GUI changes for Channel Manager with popup menus, IP Catalog, Quartus Prime, and Qsys.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added ADME and JTAG debug link info for Arria 10.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added instructions to run Tcl script from command line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added heat map display option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added procedure to use internal PLL to generate reconfig_clk.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added note stating RX CDR PLL status can toggle in LTD mode.</td>
</tr>
<tr>
<td>November, 2013</td>
<td>13.1.0</td>
<td>• Reorganization and conversion to DITA.</td>
</tr>
<tr>
<td>May, 2013</td>
<td>13.0.0</td>
<td>• Added Conduit Mode Support, Serial Bit Comparator, Required Files and Tcl command tables.</td>
</tr>
<tr>
<td>November, 2012</td>
<td>12.1.0</td>
<td>• Minor editorial updates. Added Tcl help information and removed Tcl command tables. Added 28-Gbps Transceiver support section.</td>
</tr>
<tr>
<td>August, 2012</td>
<td>12.0.1</td>
<td>• General reorganization and revised steps in modifying Altera example designs.</td>
</tr>
<tr>
<td>June, 2012</td>
<td>12.0.0</td>
<td>• Maintenance release for update of Transceiver Toolkit features.</td>
</tr>
<tr>
<td>November, 2011</td>
<td>11.1.0</td>
<td>• Maintenance release for update of Transceiver Toolkit features.</td>
</tr>
<tr>
<td>May, 2011</td>
<td>11.0.0</td>
<td>• Added new Tcl scenario.</td>
</tr>
<tr>
<td>December, 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template. Added new 10.1 release features.</td>
</tr>
<tr>
<td>August, 2010</td>
<td>10.0.1</td>
<td>• Corrected links.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Initial release.</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
Quick Design Debugging Using SignalProbe

The SignalProbe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs. Easy access to internal device signals is important in the design or debugging process. The SignalProbe feature makes design verification more efficient by routing internal signals to I/O pins quickly without affecting the design. When you start with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

The SignalProbe feature supports the Arria® series, Cyclone® series, MAX® II, and Stratix® series device families.

Related Information
- System Debugging Tools Overview documentation on page 5-1
  Overview and comparison of all the tools available in the Quartus Prime software

Design Flow Using SignalProbe

The SignalProbe feature allows you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. SignalProbe is an effective debugging tool that provides visibility into your FPGA.

You can reserve pins for SignalProbe and assign I/O standards after a full compilation. Each SignalProbe-source to SignalProbe-pin connection is implemented as an engineering change order (ECO) that is applied to your netlist after a full compilation.

To route the internal signals to the device’s reserved pins for SignalProbe, perform the following tasks:

1. Perform a full compilation.
2. Reserve SignalProbe Pins.
3. Assign SignalProbe sources.
4. Add registers between pipeline paths and Signalprobe pins.
5. Perform a SignalProbe compilation.
6. Analyze the results of a SignalProbe compilation.
Perform a Full Compilation

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe.

To perform a full compilation, on the Processing menu, click Start Compilation.

Reserve SignalProbe Pins

SignalProbe pins can only be reserved after a full compilation. You can also probe any unused I/Os of the device. Assigning sources is a simple process after reserving SignalProbe pins. The sources for SignalProbe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

Note: Although you can reserve SignalProbe pins using many features within the Quartus Prime software, including the Pin Planner and the Tcl interface, you should use the SignalProbe Pins dialog box to create and edit your SignalProbe pins.

Assign SignalProbe Sources

A SignalProbe source can be any combinational node, register, or pin in your post-compilation netlist. To find a SignalProbe source, in the Node Finder, use the SignalProbe filter to remove all sources that cannot be probed. You might not be able to find a particular internal node because the node can be optimized away during synthesis, or the node cannot be routed to the SignalProbe pin. For example, you cannot probe nodes and registers within Gigabit transceivers in Stratix IV devices because there are no physical routes available to the pins.

Because SignalProbe pins are implemented and routed as ECOs, turning the SignalProbe enable option on or off is the same as selecting Apply Selected Change or Restore Selected Change in the Change Manager window. If the Change Manager window is not visible at the bottom of your screen, on the View menu, point to Utility Windows and click Change Manager.

Related Information

- SignalProbe Pins Dialog Box online help
- Add SignalProbe Pins Dialog Box online help
- Engineering Change Management with the Chip Planner documentation

Add Registers Between Pipeline Paths and SignalProbe Pins

You can specify the number of registers placed between a SignalProbe source and a SignalProbe pin. The registers synchronize data to a clock and control the latency of the SignalProbe outputs. The SignalProbe feature automatically inserts the number of registers specified into the SignalProbe path.

The figure shows a single register between the SignalProbe source Reg_b_1 and SignalProbe SignalProbe_Output_2 output pin added to synchronize the data between the two SignalProbe output pins.

Note: When you add a register to a SignalProbe pin, the SignalProbe compilation attempts to place the register to best meet timing requirements. You can place SignalProbe registers either near the SignalProbe source to meet fMAX requirements, or near the I/O to meet tCO requirements.
In addition to clock input for pipeline registers, you can also specify a reset signal pin for pipeline registers. To specify a reset pin for pipeline registers, use the Tcl command `make_sp`.

**Related Information**

- Add SignalProbe Pins Dialog Box online help
- Information about how to pipeline an existing SignalProbe connection

**Perform a SignalProbe Compilation**

Perform a SignalProbe compilation to route your SignalProbe pins. A SignalProbe compilation saves and checks all netlist changes without recompiling the other parts of the design. A SignalProbe compilation takes a fraction of the time of a full compilation to finish. The design’s current placement and routing are preserved.

To perform a SignalProbe compilation, on the Processing menu, point to **Start** and click **Start SignalProbe Compilation**.

**Analyze the Results of a SignalProbe Compilation**

After a SignalProbe compilation, the results are available in the compilation report file. Each SignalProbe pin is displayed in the **SignalProbe Fitting Result** page in the **Fitter** section of the Compilation Report. To view the status of each SignalProbe pin in the **SignalProbe Pins** dialog box, on the Tools menu, click **SignalProbe Pins**.

The status of each SignalProbe pin appears in the Change Manager window. If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to **Utility Windows** and click **Change Manager**.
To view the timing results of each successfully routed SignalProbe pin, on the Processing menu, point to Start and click Start Timing Analysis.

Related Information
Engineering Change Management with the Chip Planner documentation

What a SignalProbe Compilation Does
After a full compilation, you can start a SignalProbe compilation either manually or automatically. A SignalProbe compilation performs the following functions:

- Validates SignalProbe pins
- Validates your specified SignalProbe sources
- Adds registers into SignalProbe paths, if applicable
- Attempts to route from SignalProbe sources through registers to SignalProbe pins

To run the SignalProbe compilation immediately after a full compilation, on the Tools menu, click SignalProbe Pins. In the SignalProbe Pins dialog box, click Start Check & Save All Netlist Changes.

To run a SignalProbe compilation manually after a full compilation, on the Processing menu, point to Start and click Start SignalProbe Compilation.

Note: You must run the Fitter before a SignalProbe compilation. The Fitter generates a list of all internal nodes that can serve as SignalProbe sources.

Turn the SignalProbe enable option on or off in the SignalProbe Pins dialog box to enable or disable each SignalProbe pin.

Understanding the Results of a SignalProbe Compilation
After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status of each SignalProbe pin appears in the SignalProbe Fitting Result screen in the Fitter section of the Compilation Report.

Table 8-1: Status Values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routed</td>
<td>Connected and routed successfully</td>
</tr>
<tr>
<td>Not Routed</td>
<td>Not enabled</td>
</tr>
<tr>
<td>Status</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Failed to Route</td>
<td>Failed routing during last SignalProbe compilation</td>
</tr>
<tr>
<td>Need to Compile</td>
<td>Assignment changed since last SignalProbe compilation</td>
</tr>
</tbody>
</table>

Figure 8-3: SignalProbe Fitting Results Page in the Compilation Report Window

The SignalProbe source to output delays screen in the Timing Analysis section of the Compilation Report displays the timing results of each successfully routed SignalProbe pin.

Figure 8-4: SignalProbe Source to Output Delays Page in the Compilation Report Window
Analyzing SignalProbe Routing Failures

A SignalProbe compilation can fail for any of the following reasons:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion.
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe source that does not exist or is invalid.
- **Unusable output pin**—the output pin selected is found to be unusable.

Routing failures can occur if the SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful SignalProbe compilation, you can allow the compiler to modify routing to the specified SignalProbe source. On the Tools menu, click *SignalProbe Pins* and turn on *Modify latest fitting results during SignalProbe compilation*. This setting allows the Fitter to modify existing routing channels used by your design.

**Note:** Turning on *Modify latest fitting results during SignalProbe compilation* can change the performance of your design.

Scripting Support

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
```

**Note:** The Tcl commands in this section are part of the *::quartus::chip_planner* Quartus Prime Tcl API. Source or include the *::quartus::chip_planner* Tcl package in your scripts to make these commands available.

Related Information

- **Tcl Scripting documentation**
- **Quartus Prime Settings File Reference Manual**
  Information about all settings and constraints in the Quartus Prime software
- **Command-Line Scripting documentation**

Making a SignalProbe Pin

To make a SignalProbe pin, type the following command:

```
```
Deleting a SignalProbe Pin

To delete a SignalProbe pin, type the following Tcl command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Enabling a SignalProbe Pin

To enable a SignalProbe pin, type the following Tcl command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Disabling a SignalProbe Pin

To disable a SignalProbe pin, type the following Tcl command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Performing a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

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

Script Example

The example shows a script that creates a SignalProbe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.

```
package require ::quartus::chip_planner
project_open project
read_netlist
make_sp -pin_name sp1 -src_name reg1
check_netlist_and_save
project_close
```

Reserving SignalProbe Pins

To reserve a SignalProbe pin, add the commands shown in the example to the Quartus Prime Settings File (.qsf) for your project.

```
set_location_assignment <location> -to <SignalProbe pin name>
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to <SignalProbe pin name>
```

Valid locations are pin location names, such as `Pin_A3`.

Common Problems When Reserving a SignalProbe Pin

If you cannot reserve a SignalProbe pin in the Quartus Prime software, it is likely that one of the following is true:
• You have selected multiple pins.
• A compilation is running in the background. Wait until the compilation is complete before reserving the pin.
• You have the Quartus Prime Lite Edition software, in which the SignalProbe feature is not enabled by default. You must turn on TalkBack to enable the SignalProbe feature in the Quartus Prime Lite Edition software.
• You have not set the pin reserve type to As Signal Probe Output. To reserve a pin, on the Assignments menu, in the Assign Pins dialog box, select As SignalProbe Output.
• The pin is reserved from a previous compilation. During a compilation, the Quartus Prime software reserves each pin on the targeted device. If you end the Quartus Prime process during a compilation, for example, with the Windows Task Manager End Process command or the UNIX kill command, perform a full recompilation before reserving pins as SignalProbe outputs.
• The pin does not support the SignalProbe feature. Select another pin.
• The current device family does not support the SignalProbe feature.

Adding SignalProbe Sources

To assign the node name to a SignalProbe pin, type the following Tcl command:

```tcl
set_instance_assignment -name SIGNALPROBE_SOURCE <node name> \
-to <SignalProbe pin name>
```

The next command turns on SignalProbe routing. To turn off individual SignalProbe pins, specify OFF instead of ON with the following command:

```tcl
set_instance_assignment -name SIGNALPROBE_ENABLE ON \
-to <SignalProbe pin name>
```

Related Information

• SignalProbe Pins Dialog Box online help
• Add SignalProbe Pins Dialog Box online help
  Information about how to pipeline an existing SignalProbe connection

Assigning I/O Standards

To assign an I/O standard to a pin, type the following Tcl command:

```tcl
set_instance_assignment -name IO_STANDARD <I/O standard> -to <SignalProbe pin name>
```

Related Information

I/O Standards online help

Adding Registers for Pipelining

To add registers for pipelining, type the following Tcl command:

```tcl
set_instance_assignment -name SIGNALPROBE_CLOCK <clock name> \ 
-to <SignalProbe pin name>
set_instance_assignment -name SIGNALPROBE_NUM_REGISTERS <number of registers> -to <SignalProbe pin name>
```
Running SignalProbe Immediately After a Full Compilation

To run SignalProbe immediately after a full compilation, type the following Tcl command:

```tcl
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

Running SignalProbe Manually

To run SignalProbe as part of a scripted flow using Tcl, use the following in your script:

```tcl
execute_flow -signalprobe
```

To perform a Signal Probe compilation interactively at a command prompt, type the following command:

```bash
quartus_sh_fit --flow signalprobe <project name>
```

Enabling or Disabling All SignalProbe Routing

Use the Tcl command in the example to turn on or turn off SignalProbe routing. When using this command, to turn SignalProbe routing on, specify `ON`. To turn SignalProbe routing off, specify `OFF`.

Turning SignalProbe On or Off with Tcl Commands

```tcl
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
    set signalprobe_pin_name [lindex $asgn 2]
    set_instance_assignment -name SIGNALPROBE_ENABLE \
    -to $signalprobe_pin_name <ON|OFF> 
}
```

Allowing SignalProbe to Modify Fitting Results

To turn on `Modify latest fitting results`, type the following Tcl command:

```tcl
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

Document Revision History

Table 8-2: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Dita conversion.</td>
</tr>
<tr>
<td>May 2013</td>
<td>13.0.0</td>
<td>Changed sequence of flow to clarify that you need to perform a full compilation before reserving SignalProbe pins. Affected sections are “Debugging Using the SignalProbe Feature” on page 12–1 and “Reserving SignalProbe Pins” on page 12–2. Moved “Performing a Full Compilation” on page 12–2 before “Reserving SignalProbe Pins” on page 12–2.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
</tbody>
</table>
## Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2011</td>
<td>10.0.2</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Revised for new UI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed section SignalProbe ECO flows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed support for SignalProbe pin preservation when recompiling with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incremental compilation turned on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed outdated FAQ section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added links to Quartus Prime Help for procedural content.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Removed all references and procedures for APEX devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Style changes.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Removed the “Generate the Programming File” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed unnecessary screenshots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>• Modified description for preserving SignalProbe connections when using</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Incremental Compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added plausible scenarios where SignalProbe connections are not reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in the design</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Added “Arria GX” to the list of supported devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed the “On-Chip Debugging Tool Comparison” and replaced with a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reference to the Section V Overview on page 13–1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
About the SignalTap II Logic Analyzer

The SignalTap® II Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in an FPGA design. You can examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA.

The SignalTap II Logic Analyzer is scalable, easy to use, and available as a stand-alone package or with a software subscription. You can debug an FPGA design by probing the state of the design’s internal signals without external equipment. Define custom trigger-condition logic for greater accuracy and improved ability to isolate problems. The SignalTap II Logic Analyzer does not require external probes or design file changes to capture the state of the internal nodes or I/O pins in the design. All captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.

The SignalTap II Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market.

Figure 9-1: SignalTap II Logic Analyzer Block Diagram

This chapter is intended for any designer who wants to debug an FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II Logic Analyzer is similar
to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful, but not necessary.

**Note:** The Quartus Prime Pro Edition software uses a new methodology for settings and assignments. For example, SignalTap II assignments include only the instance name, not the entity:instance name. Refer to Migrating to Quartus Prime Pro Edition for more information on migrating existing .stp files to Quartus Prime Pro Edition.

**Related Information**

**Migrating to Quartus Prime Pro Edition**
Migrating to the Quartus Prime Pro Edition software.

**Hardware and Software Requirements**

You need the following hardware and software to perform logic analysis with the SignalTap II Logic Analyzer:

- SignalTap software
- Download/upload cable
- Altera® development kit or your design board with JTAG connection to device under test

Use the SignalTap software that is included with the following software:

- Quartus Prime design software
- Quartus Prime Lite Edition (with the TalkBack feature enabled)

Alternatively, use the SignalTap II Logic Analyzer standalone software and standalone Programmer software.

**Note:** The Quartus Prime Lite Edition software does not support incremental compilation integration with the SignalTap II Logic Analyzer.

The memory blocks of the device store captured data. The memory blocks transfer the data to the Quartus Prime software waveform display over a JTAG communication cable, such as EthernetBlaster or USB-Blaster®.

**Table 9-1: SignalTap II Logic Analyzer Features and Benefits**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick access toolbar</td>
<td>Single-click operation of commonly used menu items. Hover over the icons to see tool tips.</td>
</tr>
<tr>
<td>Multiple logic analyzers in a single device</td>
<td>Captures data from multiple clock domains in a design at the same time.</td>
</tr>
<tr>
<td>Multiple logic analyzers in multiple devices in a single JTAG chain</td>
<td>Simultaneously captures data from multiple devices in a JTAG chain.</td>
</tr>
<tr>
<td>Nios II plug-in support</td>
<td>Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II processor.</td>
</tr>
<tr>
<td>Up to 10 basic or advanced trigger conditions for each analyzer instance</td>
<td>Enables sending more complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.</td>
</tr>
<tr>
<td>Feature</td>
<td>Benefit</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Power-up trigger</td>
<td>Captures signal data for triggers that occur after device programming, but before manually starting the logic analyzer.</td>
</tr>
<tr>
<td>Custom trigger HDL object</td>
<td>You can code your own trigger in Verilog HDL or VHDL and tap specific instances of modules located anywhere in the hierarchy of your design without needing to manually route all the necessary connections. This simplifies the process of tapping nodes spread out across your design.</td>
</tr>
<tr>
<td>State-based triggering flow</td>
<td>Enables you to organize your triggering conditions to precisely define what your logic analyzer captures.</td>
</tr>
<tr>
<td>Incremental route with rapid recompile</td>
<td>Manually allocate trigger input, data input, storage filter input node count, and perform a full compilation to include the SignalTap II Logic Analyzer in your design. Then, you can selectively connect, disconnect, and swap to different nodes in your design. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.</td>
</tr>
<tr>
<td>Flexible buffer acquisition modes</td>
<td>The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debugging of your design.</td>
</tr>
<tr>
<td>MATLAB integration with included MEX function</td>
<td>Collects the SignalTap II Logic Analyzer captured data into a MATLAB integer matrix.</td>
</tr>
<tr>
<td>Up to 2,048 channels per logic analyzer instance</td>
<td>Samples many signals and wide bus structures.</td>
</tr>
<tr>
<td>Up to 128K samples in each device</td>
<td>Captures a large sample set for each channel.</td>
</tr>
<tr>
<td>Fast clock frequencies</td>
<td>Synchronous sampling of data nodes using the same clock tree driving the logic under test.</td>
</tr>
<tr>
<td>Resource usage estimator</td>
<td>Provides estimate of logic and memory device resources used by SignalTap II Logic Analyzer configurations.</td>
</tr>
<tr>
<td>No additional cost</td>
<td>The SignalTap II Logic Analyzer is included with a Quartus Prime subscription and with the Quartus Prime Lite Edition (with TalkBack enabled).</td>
</tr>
<tr>
<td>Compatibility with other on-chip debugging utilities</td>
<td>You can use the SignalTap II Logic Analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, allowing you to change signal values in real-time while you are running an analysis with the SignalTap II Logic Analyzer.</td>
</tr>
<tr>
<td>Floating-Point Display Format</td>
<td>To enable, click Edit &gt; Bus Display Format &gt; Floating-point Supports the following:</td>
</tr>
<tr>
<td></td>
<td>- Single-precision floating-point format IEEE754 Single (32-bit)</td>
</tr>
<tr>
<td></td>
<td>- Double-precision floating-point format IEEE754 Double (64-bit)</td>
</tr>
</tbody>
</table>
Design Flow Using the SignalTap II Logic Analyzer

Add a SignalTap II file (.stp) and enable it in your project, or instantiate a SignalTap II IP core created with the parameter editor. Figure 9-2 shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to final device configuration, testing, and debugging.

Figure 9-2: SignalTap II FPGA Design and Debugging Flow

![SignalTap II FPGA Design and Debugging Flow Diagram]

SignalTap II Logic Analyzer Task Flow Overview

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer.
Add the SignalTap II Logic Analyzer to Your Design

Create an .stp or create a parameterized HDL instance representation of the logic analyzer using the IP Catalog and parameter editor. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

Configure the SignalTap II Logic Analyzer

After you add the SignalTap II Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II processor plug-in, to quickly add entire sets of associated signals for a particular intellectual property (IP). You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

Related Information
Creating a Power-Up Trigger on page 9-42
Define Trigger Conditions
The SignalTap II Logic Analyzer captures data continuously while the logic analyzer is running. To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II Logic Analyzer allows you to define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers allow you to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Compile the Design
With the .stp configured and trigger conditions defined, compile your project as usual to include the logic analyzer in your design.

Program the Target Device or Devices
When you debug a design with the SignalTap II Logic Analyzer, you can program a target device directly from the .stp without using the Quartus Prime Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Note: The SignalTap II Logic Analyzer supports all current Altera FPGA device families.

Related Information
Managing Multiple SignalTap II Files and Configurations on page 9-21

Run the SignalTap II Logic Analyzer
In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, read and transfer the captured data from the on-chip buffer to the .stp for analysis.

Related Information
Analyzing Data in the SignalTap II Logic Analyzer online help

View, Analyze, and Use Captured Data
After you have captured data and read it into the .stp, that data is available for analysis and debugging. Set up mnemonic tables, either manually or with a plug-in, to simplify reading and interpreting the captured signal data. To speed up debugging, use the Locate feature in the SignalTap II node list to find the locations of problem nodes in other tools in the Quartus Prime software. Save the captured data for later analysis, or convert the data to other formats for sharing and further study.

Embed Multiple Analyzers in One FPGA
The SignalTap II Logic Analyzer Editor includes support for adding multiple logic analyzers by creating instances in the .stp. You can create a unique logic analyzer for each clock domain in the design.

Monitor FPGA Resources Used by the SignalTap II Logic Analyzer
The SignalTap II Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each logic analyzer instance uses. Furthermore, because the most demanding on-chip resource for the logic analyzer is memory usage, the resource estimator reports the ratio of total RAM usage in your design to the total amount of RAM available, given the results of the last compilation. The resource estimator provides a warning if a potential for a “no-fit” occurs.
You can see resource usage of each logic analyzer instance and total resources used in the columns of the **Instance Manager** pane of the SignalTap II Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value reported in the resource usage estimator may vary by as much as 10% from the actual resource usage.

**Use the Parameter Editor to Create Your Logic Analyzer**

You can create a SignalTap II Logic Analyzer instance by using the parameter editor. The parameter editor generates an HDL file that you instantiate in your design.

**Note:** The state-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the parameter editor to create the logic analyzer.

**Configuring the SignalTap II Logic Analyzer**

There are several ways to configure instances of the SignalTap II Logic Analyzer. Some settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Logic Analyzer because it is configurable.

**Note:** You can only adjust some settings when you are viewing run-time trigger conditions instead of power-up trigger conditions.

**Assigning an Acquisition Clock**

Assign a clock signal to control how the SignalTap II Logic Analyzer acquires data. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock. You can use any signal in your design as the acquisition clock. However, for best results, Altera recommends that you use a global, non-gated clock synchronous to the signals under test for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus Prime static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. Refer to the Timing Analysis section of the Compilation Report to find the maximum frequency of the logic analyzer clock.

**Note:** Exercise caution when using a recovered clock from a transceiver as an acquisition clock for the SignalTap II Logic Analyzer. A recovered clock can cause incorrect or unexpected behavior, particularly when the transceiver recovered clock is the acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the SignalTap II Logic Analyzer Editor, the Quartus Prime software automatically creates a clock pin called `auto_stp_external_clk`. You must make a pin assignment to this pin and ensure that a clock signal in your design drives the acquisition clock.

**Related Information**

-[Managing Device I/O Pins](#)

Information about assigning signals to pins

**Adding Signals to the SignalTap II File**

Add signals to the `.stp` node list to select which signals in your design you want to monitor. You can also select signals to define triggers. You can assign the following two signal types:
• **Pre-synthesis**—These signals exist after design elaboration, but before any synthesis optimizations are done. This set of signals should reflect your Register Transfer Level (RTL) signals.

• **Post-fitting**—This signal exists after physical synthesis optimizations and place-and-route.

The Quartus Prime software does not limit the number of signals available for monitoring in the SignalTap II window waveform display. However, the number of channels available is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

**Note:** The Quartus Prime Pro Edition software uses only the instance name, and not the entity name, in the form of:

\[ \text{a|b|c} \]

\[ \text{not a_entity:a|b_entity:b|c_entity:c} \]

After successful Analysis and Elaboration, invalid signals are displayed in red. Unless you are certain that these signals are valid, remove them from the .stp for correct operation. The SignalTap II Status Indicator also indicates if an invalid node name exists in the .stp.

You can tap signals if a routing resource (row or column interconnects) exists to route the connection to the SignalTap II instance. For example, signals that exist in the I/O element (IOE) cannot be directly tapped because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can tap the signal that is driving a logic array block (LAB) from an IOE, or, for output pins, you can tap the signal from the LAB that is driving an IOE.

When adding pre-synthesis signals, make all connections to the SignalTap II Logic Analyzer before synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. Pre-synthesis signal names for signals driving to and from IOEs coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. You can only make a connection if the signals are part of the existing post-fit netlist and existing routing resources are available from the signal of interest to the SignalTap II Logic Analyzer. In the case of post-fit output signals, tap the COMBOUT or REGOUT signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the signal name assigned to the pin.

**Note:** Because NOT-gate push back applies to any register that you tap, the signal from the atom may be inverted. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. The Technology Map viewer and the Resource Property Editor can also be used to help you find post-fit node names.

**Related Information**

Analyzing Designs with Quartus Prime Netlist Viewers documentation
Information about cross-probing to source design files and other Quartus Prime windows

**Preserving Signals**

The Quartus Prime software optimizes the RTL signals during synthesis and place-and-route. RTL signal names frequently may not appear in the post-fit netlist after optimizations.
For example, the compilation process can add tildes (~) to nets that fan-out from a node, making it difficult to decipher which signal nets they actually represent. When the Quartus Prime software encounters the following synthesis attributes, it does not perform any optimization on the specified signals, allowing them to persist into the post-fit netlist.

- keep—Ensures that combinational signals are not removed
- preserve—Ensures that registers are not removed

Using these attributes can increase device resource utilization or decrease timing performance.

If you are debugging an IP core, such as the Nios II CPU or other encrypted IP, you might need to preserve nodes from the core to make them available for debugging with the SignalTap II Logic Analyzer. Preserving nodes is often necessary when a plug-in is used to add a group of signals for a particular IP.

Related Information
Quartus Prime Integrated Synthesis documentation
Information about using signal preservation attributes

Assigning Data Signals Using the Technology Map Viewer
You can use the Technology map viewer to add post-fit signal names easily. To do so, launch the Technology map viewer (post-fitting) after compiling your design. When you find the desired node, copy the node to either the active .stp for your design or a new .stp.

Node List Signal Use Options
When you add a signal to the node list, you can select options that specify how the logic analyzer uses the signal.

You can turn off the ability of a signal to trigger the analyzer by disabling the Trigger Enable option for that signal in the node list in the .stp. This option is useful when you want to see only the captured data for a signal and you are not using that signal as part of a trigger.

You can turn off the ability to view data for a signal by disabling the Data Enable column. This option is useful when you want to trigger on a signal, but have no interest in viewing data for that signal.

Related Information
Define Triggers on page 9-22

Disabling and Enabling a SignalTap II Instance
Disable and enable SignalTap II instances in the Instance Manager pane. Physically adding or removing instances requires recompilation after disabling and enabling a SignalTap II instance.

Untappable Signals
Not all of the post-fitting signals in your design are available in the SignalTap II : post-fitting filter in the Node Finder dialog box. The following signal types cannot be tapped:
• **Post-fit output pins**—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin. This includes pins defined as bidirectional.

• **Signals that are part of a carry chain**—You cannot tap the carry out \((\text{cout}_0 \text{ or } \text{cout}_1)\) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another LE.

• **JTAG Signals**—You cannot tap the JTAG control \((\text{TCK}, \text{TDI}, \text{TDO}, \text{and } \text{TMS})\) signals.

• **ALTGXB IP core**—You cannot directly tap any ports of an ALTGXB instantiation.

• **LVDS**—You cannot tap the data output from a serializer/deserializer (SERDES) block.

• **DQ, DQS Signals**—You cannot directly tap the \(\text{DQ}\) or \(\text{DQS}\) signals in a DDR/DDRII design.

### Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can add groups of relevant signals of a particular type of IP with a plug-in. The SignalTap II Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also provide features such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data.

The Nios II plug-in, for example, creates one mnemonic table in the **Setup** tab and two tables in the **Data** tab:

- **Nios II Instruction** (**Setup** tab)—Capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address** (**Data** tab)—Display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (**.elf**) file.
- **Nios II Disassembly** (**Data** tab)—Displays disassembled code from the corresponding address.

To add signals to the **.stp** using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. Right-click in the node list. On the Add Nodes with Plug-In submenu, choose the plug-in you want to use, such as the included plug-in named **Nios II**.

   **Note:** If the IP for the selected plug-in does not exist in your design, a message informs you that you cannot use the selected plug-in.

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.

3. If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in selected, where you can specify options for the plug-in. With the Nios II plug-in, you can optionally select an **.elf** containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify options for the selected plug-in as desired and click **OK**.

   **Note:** To make sure all the required signals are available, in the Quartus Prime **Analysis & Synthesis** settings, click **Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis)**. Turn on **Create debugging nodes for IP cores**.

All the signals included in the plug-in are added to the node list.

### Related Information

- **Define Triggers** on page 9-22
- **View, Analyze, and Use Captured Data** on page 9-6
Adding Finite State Machine State Encoding Registers

Finding the signals to debug finite state machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you can find all of the relevant nodes in the post-fit netlist or you used the nodes from the pre-synthesis netlist, an additional step is required to find and map FSM signal values to the state names that you specified in your HDL.

The SignalTap II Logic Analyzer can detect FSMs in your compiled design. The SignalTap II Logic Analyzer configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Shortcut menu commands allow you to add all of the FSM state signals to your logic analyzer with a single command. For each FSM added to your SignalTap II configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 9-4: Decoded FSM Mnemonics

The waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>16</th>
<th>8</th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related Information

Recommended HDL Coding Styles documentation
Coding guidelines for specifying FSM in Verilog HDL and VHDL

Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II Logic Analyzer GUI creates a mnemonic table using the format $<StateSignalName>_table$, where $StateSignalName$ is the name of the state signals that you have declared in your RTL. You can edit any mnemonic table using the Mnemonic Table Setup dialog box.

If you want to restore a mnemonic table that was modified, right-click anywhere in the node list window and select Recreate State Machine Mnemonics. By default, restoring a mnemonic table overwrites the existing mnemonic table that you modified. To restore a FSM mnemonic table to a new record, turn off Overwrite existing mnemonic table in the Recreate State Machine Mnemonics dialog box.

Note: If you have added or deleted a signal from the FSM state signal group from within the setup tab, delete the modified register group and add the FSM signals back again.

Related Information

Creating Mnemonics for Bit Patterns on page 9-54

Additional Considerations

The SignalTap II configuration GUI recognizes state machines from your design only if you use Quartus Prime Integrated Synthesis (QIS). The state machine debugging feature is not able to track the FSM signals or state encoding if you use other EDA synthesis tools.
If you add post-fit FSM signals, the SignalTap II Logic Analyzer FSM debug feature may not track all optimization changes that are a part of the compilation process. If the following two specific optimizations are enabled, the SignalTap II FSM debug feature may not list mnemonic tables for state machines in the design:

- If you have physical synthesis turned on, state registers may be resource balanced (register retiming) to improve $f_{\text{MAX}}$. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
- The FSM debugging feature does not list state signals that have been packed into RAM and DSP blocks during QIS or Fitter optimizations.

You can still use the FSM debugging feature to add pre-synthesis state signals.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To specify the sample depth, select the desired number of samples to store in the Sample Depth list. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.

Capturing Data to a Specific RAM Type

When you use the SignalTap II Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. Once SignalTap II Logic Analyzer is allocated to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II Logic Analyzer data acquisition. For example, if your design has an application that requires a large block of memory resources, such as a large instruction or data cache, you would choose to use MLAB, M512, or M4k blocks for data acquisition and leave the M9k blocks for the rest of your design.

To select the RAM type to use for the SignalTap II Logic Analyzer buffer, select it from the RAM type list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. There are two types of acquisition buffer within the SignalTap II Logic Analyzer—a non-segmented (or circular) buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the logic analyzer reaches a defined set of trigger conditions. With a segmented buffer, the memory space is split into a number of separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions. Only a single buffer is active during an acquisition. The SignalTap II Logic Analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space. Figure 9-5 illustrates the differences between the two buffer types.
Figure 9-5: Buffer Type Comparison in the SignalTap II Logic Analyzer

Notes to figure:

1. Both non-segmented and segmented buffers can use a predefined trigger (Pre-Trigger, Center Trigger, Post-Trigger) position or define a custom trigger position using the State-Based Triggering tab. Refer to Specifying the Trigger Position for more details.
2. Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).

Related Information
Using the Storage Qualifier Feature on page 9-14

Non-Segmented Buffer

The non-segmented buffer (also known as a circular buffer) shown in Figure 9-5 (a) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event, consisting of a set of trigger conditions, occurs. When the trigger event happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the trigger position setting in the Signal Configuration pane or the .stp. To capture the majority of the data before the trigger occurs, select Post trigger position from the list. To capture the majority of the data after the trigger, select Pre-trigger position. To center the trigger position in the data, select Center trigger position. Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

Related Information
Specifying the Trigger Position on page 9-41

Segmented Buffer

A segmented buffer allows you to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow. Figure 9-6 shows an example of a segmented buffer system. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow.
The SignalTap II Logic Analyzer verifies the functionality of the design shown in Figure 9-6 to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Logic Analyzer allows you to monitor the RDATA port when $H'0F0F0F0F$ is sent into the RADDR port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so you can capture the same event multiple times without wasting allocated memory. The number of cycles that are captured depends on the number of segments specified under the Data settings.

To enable and configure buffer acquisition, select Segmented in the SignalTap II Logic Analyzer Editor and select the number of segments to use. In the example in Figure 9-6, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the RADDR signal is $H'0F0F0F0F$.

### Using the Storage Qualifier Feature

Both non-segmented and segmented buffers described in the previous section offer a snapshot in time of the data stream being analyzed. The default behavior for writing into acquisition memory with the SignalTap II Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Logic Analyzer, you cannot discard the redundant sample bits.

The Storage Qualification feature allows you to filter out individual samples not relevant to debugging the design. With this feature, a condition acts as a write enable to the buffer during each clock cycle of data acquisition. Through fine tuning the data that is actually stored in acquisition memory, the Storage Qualification feature allows for a more efficient use of acquisition memory in the specified number of samples over a longer period of analysis.

Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two
samples in the buffer, the Storage Qualification feature is equivalent to being able to create a customized segmented buffer in which the number and size of segment boundaries are adjustable.

**Note:** You can only use the Storage Qualification feature with a non-segmented buffer. The IP Catalog flow only supports the Input Port mode for the Storage Qualification feature.

**Figure 9-7: Data Acquisition Using Different Modes of Controlling the Acquisition Buffer**

Notes to figure:

1. Non-segmented Buffers capture a fixed sample window of contiguous data.
2. Segmented buffers divide the buffer into fixed sized segments, with each segment having an equal sample depth.
3. Storage Qualification allows you to define a custom sampling window for each segment you create with a qualifying condition. Storage qualification potentially allows for a larger time scale of coverage.
There are six storage qualifier types available under the Storage Qualification feature:

- Continuous
- Input port
- Transitional
- Conditional
- Start/Stop
- State-based

Continuous (the default mode selected) turns the Storage Qualification feature off.

Each selected storage qualifier type is active when an acquisition starts. Upon the start of an acquisition, the SignalTap II Logic Analyzer examines each clock cycle and writes the data into the acquisition buffer based upon storage qualifier type and condition. The acquisition stops when a defined set of trigger conditions occur.

**Note:** Trigger conditions are evaluated independently of storage qualifier conditions. The SignalTap II Logic Analyzer evaluates the data stream for trigger conditions on every clock cycle after the acquisition begins.

The storage qualifier operates independently of the trigger conditions.

**Related Information**

Define Trigger Conditions on page 9-6

**Input Port Mode**

When using the Input port mode, the SignalTap II Logic Analyzer takes any signal from your design as an input. When the design is running, if the signal is high on the clock edge, the SignalTap II Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the data sample is ignored. A pin is created and connected to this input port by default if no internal node is specified.

If you are using an .stp to create a SignalTap II Logic Analyzer instance, specify the storage qualifier signal using the input port field located on the Setup tab. You must specify this port for your project to compile.

If you use the parameter editor, the storage qualification input port, if specified, appears in the generated instantiation template. You can then connect this port to a signal in your RTL.

**Figure 9-8: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (Segmented Buffer)**
Transitional Mode

In Transitional mode, you choose a set of signals for inspection using the node list check boxes in the Storage Qualifier column. During acquisition, if any of the signals marked for inspection have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals marked have changed since the previous clock cycle, no data is stored.

Figure 9-10: Transitional Storage Qualifier Setup

Figure 9-11: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (Transitional Mode)
Conditional Mode

In Conditional mode, the SignalTap II Logic Analyzer evaluates a combinational function of storage qualifier enabled signals within the node list to determine whether a sample is stored. The SignalTap II Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

You can select either Basic AND, Basic OR, or Advanced storage qualifier conditions. A Basic AND or Basic OR storage qualifier condition matches each signal to one of the following:

- Don’t Care
- Low
- High
- Falling Edge
- Rising Edge
- Either Edge

If you specify a Basic AND storage qualifier condition for more than one signal, the SignalTap II Logic Analyzer evaluates the logical AND of the conditions.

Any other combinational or relational operators that you may want to specify with the enabled signal set for storage qualification can be done with an advanced storage condition. Figure 9-13 details the conditional storage qualifier setup in the .stp.

You can specify storage qualification conditions similar to the manner in which trigger conditions are specified.

Figure 9-14 and Figure 9-15 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.
Start/Stop Mode

The Start/Stop mode is similar to the Conditional mode for storage qualification. However, in this mode there are two sets of conditions, one for start and one for stop. If the start condition evaluates to `TRUE`, data is stored in the buffer every clock cycle until the stop condition evaluates to `TRUE`, which then pauses.
the data capture. Additional start signals received after the data capture has started are ignored. If both start and stop evaluate to `TRUE` at the same time, a single cycle is captured.

**Note:** You can force a trigger by pressing the **Stop** button if the buffer fails to fill to completion due to a stop condition.

Figure 9-16 shows the Start/Stop mode storage qualifier setup. Figure 9-17 and Figure 9-18 show a data capture pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

**Figure 9-16: Start/Stop Mode Storage Qualifier Setup**

![Start/Stop Mode Storage Qualifier Setup](image1)

**Figure 9-17: Data Acquisition of a Recurring Data Pattern in Continuous Mode (Start/Stop)**

![Data Acquisition of a Recurring Data Pattern in Continuous Mode (Start/Stop)](image2)

**Figure 9-18: Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled**

![Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled](image3)

**State-Based**

The State-based storage qualification mode is used with the State-based triggering flow. The state based triggering flow evaluates an if-else based language to define how data is written into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution
flow for the target acquisition buffer. When the storage qualifier feature is enabled for the State-based flow, two additional commands are available, the `start_store` and `stop_store` commands. These commands operate similarly to the Start/Stop capture conditions described in the previous section. Upon the start of acquisition, data is not written into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions are performed within the same clock cycle, a single sample is stored into the acquisition buffer.

**Related Information**

*State-Based Triggering* on page 9-32

**Showing Data Discontinuities**

When you turn on *Record data discontinuities*, the SignalTap II Logic Analyzer marks the samples during which the acquisition paused from a storage qualifier. This marker is displayed in the waveform viewer after acquisition completes.

**Disable Storage Qualifier**

You can turn off the storage qualifier quickly with the *Disable Storage Qualifier* option, and perform a continuous capture. This option is run-time reconfigurable; that is, the setting can be changed without recompiling the project. Changing storage qualifier mode from the *Type* field requires a recompilation of the project.

**Related Information**

*Runtime Reconfigurable Options* on page 9-48

**Managing Multiple SignalTap II Files and Configurations**

You can use more than one .stp in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals can also be used to define different sets of trigger conditions. Along with each .stp, there is also an associated programming file (SRAM Object File [.sof]). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated .sof for the logic analyzer to run properly when the device is programmed. Use the Data Log feature and the SOF Manager to manage all of the .stp files and their associated settings and programming files.

The Data Log allows you to store multiple SignalTap II configurations within a single .stp. *Figure 9-19* shows two signal set configurations with multiple trigger conditions in one .stp. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the *Setup* tab of the .stp. The active configuration displayed in the .stp is indicated by the blue square around the signal specified in the Data Log. Enable the Data Log by clicking the check box next to *Data Log*. To store a configuration in the Data Log, on the Edit menu, click *Save to Data Log* or click the *Save to Data Log* icon at the top of the Data Log. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.
The SOF Manager allows you to embed multiple SOFs into one .stp. Embedding an SOF in an .stp lets you move the .stp to a different location, either on the same computer or across a network, without the need to include the associated .sof separately. To embed a new SOF in the .stp, right-click in the SOF Manager, and click Attach SOF File.

As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration. You can use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA, ensuring that the configuration of your .stp always matches the design programmed into the target device.

**Define Triggers**

When you start the SignalTap II Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Logic Analyzer “triggers”—that is, the logic analyzer stops and displays the data—when a condition or set of conditions that you specified has been reached. This section describes the various types of trigger conditions that you can specify using the SignalTap II Logic Analyzer on the Signal Configuration pane.
Creating Basic Trigger Conditions

The simplest kind of trigger condition is a basic trigger. Select this from the list at the top of the Trigger Conditions column in the node list in the SignalTap II Logic Analyzer Editor. If you select the Basic AND or Basic OR trigger type, you must specify the trigger pattern for each signal you have added in the .stp. To specify the trigger pattern, right-click in the Trigger Conditions column and click the desired pattern. Set the trigger pattern to any of the following conditions:

- Don’t Care
- Low
- High
- Falling Edge
- Rising Edge
- Either Edge

For buses, type a pattern in binary, or right-click and select Insert Value to enter the pattern in other number formats. Note that you can enter x to specify a set of “don’t care” values in either your hexadecimal or your binary string. For signals added to the .stp that have an associated mnemonic table, you can right-click and select an entry from the table to specify pre-defined conditions for the trigger.

For more information about creating and using mnemonic tables, refer to View, Analyze, and Use Captured Data, and to the Quartus Prime Help.

For signals added with certain plug-ins, you can create basic triggers easily using predefined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an .elf from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the specified code function name.

Data capture stops and the data is stored in the buffer when the logical AND of all the signals for a given trigger condition evaluates to TRUE.

Using the Basic OR Triggering Condition with Nested Groups

When you specify a set of signals as a nested group (group of groups) with the Basic OR trigger type, an advanced trigger condition is generated. This advanced trigger condition sorts signals within groups to minimize the need to recompile your design. As long as the parent-child relationships of nodes are kept constant, the generated advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design. The precedence of how this trigger condition is evaluated starts at the bottom-level with the leaf-groups first, then their resulting logic 1 or logic 0 value is used to compute the result of their parent group’s logic value. Specifying a value of TRUE for a group sets that group’s logical result to logic 1 and effectively eliminates all members beneath it from affecting the result of the group trigger. Specifying a value of FALSE for a group sets that group’s logical result to logic-0 and effectively eliminates all members beneath it from affecting the result of the group trigger.

1. Select Basic OR under Trigger Conditions.
2. In the Setup tab, select nodes including groups.
3. Right-click in the Setup tab and select Group.
4. Select your signal(s) and right-click to set a group trigger condition that applies the reduction AND, OR, NAND, NOR, XOR, XNOR, or logical TRUE or FALSE.

Note: The OR and AND group trigger conditions are only selectable for groups with no groups as children (bottom-level groups).
Figure 9-21: Creating Nested Groups
Creating Advanced Trigger Conditions

With the basic triggering capabilities of the SignalTap II Logic Analyzer, you can build more complex triggers with extra logic that enables you to capture data when a combination of conditions exist. If you select the Advanced trigger type at the top of the Trigger Conditions column in the node list of the SignalTap II Logic Analyzer Editor, a new tab named Advanced Trigger appears where you can build a complex trigger expression using a simple GUI. Drag-and-drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. To configure the operators' settings, double-click or right-click the operators that you have placed and select Properties.

Table 9-2: Advanced Triggering Operators

<table>
<thead>
<tr>
<th>Name of Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Less Than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Equality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Inequality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater Than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Logical NOT</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical OR</td>
<td>Logical</td>
</tr>
</tbody>
</table>
### Examples of Advanced Triggering Expressions

**Name of Operator** | **Type**
--- | ---
Logical XOR | Logical
Reduction AND | Reduction
Reduction OR | Reduction
Reduction XOR | Reduction
Left Shift | Shift
Right Shift | Shift
Bitwise Complement | Bitwise
Bitwise AND | Bitwise
Bitwise OR | Bitwise
Bitwise XOR | Bitwise
Edge and Level Detector | Signal Detection

**Note to table:**

1. For more information about each of these operators, refer to the Quartus Prime Help.

Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition Editor window.

### Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

**Figure 9-23: Bus outa Is Greater Than or Equal to Bus outb**

Trigger when bus \( \text{outa} \) is greater than or equal to \( \text{outb} \).
Figure 9-24: Enable Signal Has a Rising Edge

Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge.

Figure 9-25: Bitwise AND Operation

Trigger when bus `outa` is greater than or equal to bus `outb`, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed between bus `outc` and bus `outd`, and all bits of the result of that operation are equal to 1.

Custom Trigger HDL Object

The Custom Trigger HDL object found in the Advanced Trigger editor allows you to create a customized trigger condition with your own HDL module in either Verilog or VHDL. You can use this object to simulate the behavior of your triggering logic to make sure that the logic itself is not faulty. You can tap specific instances of modules located anywhere in the hierarchy of your design without having to manually route all the necessary connections.
Custom Trigger Flow

1. Select Advanced for a given trigger-level to make the Advanced Trigger editor active.
2. Prepare your Custom Trigger HDL module. You can either add a new source file to Quartus Prime that contains the trigger module or append the HDL for your trigger module to a source file already included in Quartus Prime Files under the Project Navigator.

3. Implement the required inputs and outputs for your Custom Trigger HDL module, see Table 9-3.
4. Drag in your Custom Trigger HDL object and connect the object’s data input bus and result output bit to the final trigger result.
5. Right-click your Custom Trigger HDL object and configure the object’s properties, see Table 9-4.

**Figure 9-29: Configure Object Properties**

6. Compile your design.
7. Acquire data with SignalTap II using your custom Trigger HDL object.

### Table 9-3: Custom Trigger HDL Module Required Inputs and Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Input/Output</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>acq_clk</td>
<td>Acquisition clock used by SignalTap II</td>
<td>Input</td>
<td>Required</td>
</tr>
<tr>
<td>reset</td>
<td>Reset signal used by SignalTap II when restarting a capture.</td>
<td>Input</td>
<td>Required</td>
</tr>
</tbody>
</table>
| data_in  | • Data input to be connected in the Advanced Trigger editor.  
           • Data your module will use to trigger.                      | Input        | Required          |
| pattern_in| • Module’s input for the configuration bitstream property.  
           • Runtime configurable property that can be set from SignalTap II GUI to change the behavior of your trigger logic. | Input        | Optional          |
| trigger_out| Output signal of your module to be asserted when triggering conditions have been met. | Output       | Required          |

### Table 9-4: Custom Trigger HDL Module Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom HDL Module Name</td>
<td>Module name of your triggering logic i.e. module <strong>trigger.foo</strong> (input x, y ...);</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Configuration Bitstream</td>
<td>• Allows you to create runtime-configurable trigger logic which can change its behavior based upon the value of the configuration bitstream.</td>
</tr>
<tr>
<td></td>
<td>• Configuration bitstream property is interpreted as binary and should only contain the characters 1 and 0. The bit-width (number of 1s and 0s) should match the pattern_in bit width in Table 9-3.</td>
</tr>
<tr>
<td></td>
<td>• A blank configuration bitstream implies that your module does not have a pattern_in input.</td>
</tr>
<tr>
<td>Pipeline</td>
<td>• Tells the advanced trigger editor how many stages of pipeline your triggering logic has.</td>
</tr>
<tr>
<td></td>
<td>• If it takes three clock cycles after a triggering input is received for the trigger output to be asserted, you can denote a pipeline value of three.</td>
</tr>
</tbody>
</table>

**Figure 9-30: Example of Verilog Trigger Using Configuration Bitstream**

```verilog
test_trigger(input acq_clk, reset, input [3:0] data_in, input [1:0] pattern_in, output reg trigger_out);
always @ (pattern_in) begin
  case (pattern_in)
    2'b00:
      trigger_out = &data_in;
    2'b01:
      trigger_out = |data_in;
    2'b10:
      trigger_out = 1'b0;
    2'b11:
      trigger_out = 1'b1;
  endcase
end
endmodule
```

**Figure 9-31: Example of Verilog Trigger with No Configuration Bitstream**

```verilog
test_trigger_no_bs(input acq_clk, reset, input [3:0] data_in, output trigger_out);
assign trigger_out = &data_in;
endmodule
```

**Trigger Condition Flow Control**
The SignalTap II Logic Analyzer offers multiple triggering conditions to give you precise control of the method in which data is captured into the acquisition buffers. Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. The SignalTap II Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:
• **Sequential Triggering**—The default triggering flow. Sequential triggering allows you to define up to 10 triggering levels that must be satisfied before the acquisition buffer finishes capturing.

• **State-Based Triggering**—Allows you the greatest control over your acquisition buffer. Custom-based triggering allows you to organize trigger conditions into states based on a conditional flow that you define.

You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

### Sequential Triggering
Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to `TRUE`, the SignalTap II Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. Use the Simple Sequential Triggering feature with basic triggers, advanced triggers, or a mix of both. **Figure 9-32** illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

**Note:** The external trigger is considered as trigger level 0. The external trigger must be evaluated before the main trigger levels are evaluated.

**Figure 9-32: Sequential Triggering Flow**

![Sequential Triggering Flow Diagram]

Notes to figure:

1. The acquisition buffer stops capture when all n triggering levels are satisfied, where $n \leq 10$.
2. An external trigger input, if defined, is evaluated before all other defined trigger conditions are evaluated.

To configure the SignalTap II Logic Analyzer for Sequential triggering, in the SignalTap II editor on the *Trigger flow control* list, select *Sequential*. Select the desired number of trigger conditions from the *Trigger Conditions* list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, turn on the trigger condition at the top of the column in the node list.
State-Based Triggering

Custom State-based triggering provides the most control over triggering condition arrangement. The State-Based Triggering flow allows you to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom State-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.

Events that trigger the acquisition buffer are organized by a state diagram that you define. All actions performed by the acquisition buffer are captured by the states and all transition conditions between the states are defined by the conditional expressions that you specify within each state.

Figure 9-33: State-Based Triggering Flow

Notes to figure:

1. You are allowed up to 20 different states.
2. An external trigger input, if defined, is evaluated before any conditions in the custom State-based triggering flow are evaluated.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent on a combination of triggering conditions (configured within the Setup tab), counters, and status flags. Counters and status flags are resources provided by the SignalTap II Logic Analyzer custom-based triggering flow.

Within each conditional expression you define a set of actions. Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples captured before stopping acquisition of the current segment. The count argument allows you to control the amount of data captured precisely before and after triggering event.

Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The counter and status flag resources are used as optional inputs in conditional expressions.
Counters and status flags are useful for counting the number of occurrences of particular events and for aiding in triggering flow control.

This SignalTap II custom State-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, select **State-based** on the Trigger Flow Control list. (Note that when Trigger Flow Control is specified as Sequential, the State-Based Trigger Flow tab is hidden.)

The **State-Based Trigger Flow** tab is partitioned into the following three panes:

- State Diagram pane
- Resources pane
- State Machine pane

**State Diagram Pane**

The **State Diagram** pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between the states. You can adjust the number of available states by using the menu above the graphical overview.

**State Machine Pane**

The **State Machine** pane contains the text entry boxes where you can define the triggering flow and actions associated with each state. You can define the triggering flow using the SignalTap II Trigger Flow Description Language, a simple language based on “if-else” conditional statements. Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes. The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

The State Machine description text boxes default to show one text box per state. You can also have the entire flow description shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode**.

**Related Information**

SignalTap II Trigger Flow Description Language on page 9-34

**Resources Pane**

The **Resources** pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can specify up to 20 counters and 20 status flags. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective pull-down list, and selecting **Set Initial Value**. To specify a counter width, right-click the counter name and select **Set Width**. Counters and flag values are updated dynamically after acquisition has started to assist in debugging your trigger flow specification.

The **configurable at runtime** options in the **Resources** pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation.
Table 9-5: Runtime Reconfigurable Settings, State-Based Triggering Flow

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination of goto action</td>
<td>Allows you to modify the destination of the state transition at runtime.</td>
</tr>
<tr>
<td>Comparison values</td>
<td>Allows you to modify comparison values in Boolean expressions at runtime. In</td>
</tr>
<tr>
<td></td>
<td>addition, you can modify the segment_trigger and trigger action post-fill count</td>
</tr>
<tr>
<td></td>
<td>argument at runtime.</td>
</tr>
<tr>
<td>Comparison operators</td>
<td>Allows you to modify the operators in Boolean expressions at runtime.</td>
</tr>
<tr>
<td>Logical operators</td>
<td>Allows you to modify the logical operators in Boolean expressions at runtime.</td>
</tr>
</tbody>
</table>

You can restrict changes to your SignalTap II configuration to include only the options that do not require a recompilation. Trigger lock-mode allows you to make changes that can be immediately reflected in the device.

1. On the Setup tab, select **Allow trigger condition changes only**.
2. Modify the Trigger Flow conditions in the **Custom Trigger Flow** tab.
3. Click the desired parameter in the text box and select a new parameter from the menu that appears.

**Note:** Trigger lock mode restricts changes to the configuration settings that have **configurable at runtime** specified. The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options.

**Related Information**
- Performance and Resource Considerations on page 9-45
- Runtime Reconfigurable Options on page 9-48

**SignalTap II Trigger Flow Description Language**

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions. Each line in the example shows a language format. Keywords are shown in bold. Non-terminals are delimited by “<” and are further explained in the following sections. Optional arguments are delimited by “[]”.

```
state <State_label>:
    <action_list>

if( <Boolean_expression> )
    <action_list>
[else if ( <boolean_expression> )
    <action_list>]
[else
    <action_list>]
```

**Note:** Multiple else if conditions are allowed.

The priority for evaluation of conditional statements is assigned from top to bottom. The `<boolean_expression>` in an if statement can contain a single event, or it can contain multiple event conditions. The action_list within an if or an else if clause must be delimited by the begin and end tokens when the action list contains multiple statements. When the boolean expression is evaluated **TRUE**,
the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer
- Manipulating a counter or status flag resource
- Defining a state transition

**Related Information**

*Custom Triggering Flow Application Examples* on page 9-61

**State Labels**

State labels are identifiers that can be used in the action `goto`.

`state <state_label>`: begins the description of the actions evaluated when this state is reached.

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

**Boolean_expression**

*Boolean_expression* is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.

Logical operators accept any boolean expression as an operand.

**Table 9-6: Logical Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT operator</td>
<td>! expr1</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND operator</td>
<td>expr1 &amp;&amp; expr2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relational operators are performed on counters or status flags. The comparison value, the right operator, must be a numerical value.

**Table 9-7: Relational Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&lt;identifier&gt; &gt; &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal to</td>
<td>&lt;identifier&gt; &gt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
<td>&lt;identifier&gt; == &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>!=</td>
<td>Does not equal</td>
<td>&lt;identifier&gt; != &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Syntax</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td><code>&lt;identifier&gt; &lt;= &lt;numerical_value&gt;</code></td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td><code>&lt;identifier&gt; &lt; &lt;numerical_value&gt;</code></td>
</tr>
</tbody>
</table>

Notes to table:
1. `<identifier>` indicates a counter or status flag.
2. `<numerical_value>` indicates an integer.

**Action_list**

*Action_list* is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by `begin` and `end`. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (`;`).

**Resource Manipulation Action**

The resources used in the trigger flow description can be either counters or status flags.

**Table 9-8: Resource Manipulation Action**

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>Increments a counter resource by 1</td>
<td><code>increment &lt;counter_identifier&gt;;</code></td>
</tr>
<tr>
<td>decrement</td>
<td>Decrements a counter resource by 1</td>
<td><code>decrement &lt;counter_identifier&gt;;</code></td>
</tr>
<tr>
<td>reset</td>
<td>Resets counter resource to initial value</td>
<td><code>reset &lt;counter_identifier&gt;;</code></td>
</tr>
<tr>
<td>set</td>
<td>Sets a status Flag to 1</td>
<td><code>set &lt;register_flag_identifier&gt;;</code></td>
</tr>
<tr>
<td>clear</td>
<td>Sets a status Flag to 0</td>
<td><code>clear &lt;register_flag_identifier&gt;;</code></td>
</tr>
</tbody>
</table>

**Buffer Control Action**

Buffer control actions specify an action to control the acquisition buffer.

**Table 9-9: Buffer Control Action**

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger</td>
<td>Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.</td>
<td><code>trigger &lt;post-fill_count&gt;;</code></td>
</tr>
<tr>
<td>Action</td>
<td>Description</td>
<td>Syntax</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
</tbody>
</table>
| segment_trigger | Ends the acquisition of the current segment. The SignalTap II Logic Analyzer starts acquiring from the next segment on evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode. | segment_trigger <post-fill_count>;
| start_store     | Asserts the write_enable to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.                                                        | start_store                   |
| stop_store      | De-asserts the write_enable signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.                                                        | stop_store                    |

Both `trigger` and `segment_trigger` actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the **Setup** tab.

**Note:** In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

**State Transition Action**

The State Transition action specifies the next state in the custom state control flow. It is specified by the `goto` command. The syntax is as follows:

```
goto <state_label>;
```

**Using the State-Based Storage Qualifier Feature**

When you select State-based for the storage qualifier type, the `start_store` and `stop_store` actions are enabled in the State-based trigger flow. These commands, when used in conjunction with the expressions of the State-based trigger flow, give you maximum flexibility to control data written into the acquisition buffer.

**Note:** The `start_store` and `stop_store` commands can only be applied to a non-segmented buffer.

The `start_store` and `stop_store` commands function similar to the start and stop conditions when using the `start/stop` storage qualifier mode conditions. If storage qualification is enabled, the `start_store` command must be issued for SignalTap II to write data into the acquisition buffer. No data is acquired until the `start_store` command is performed. Also, a `trigger` command must be included as part of the trigger flow description. The `trigger` command is necessary to complete the acquisition and display the results on the waveform display.
The following example illustrates the behavior of the State-based trigger flow with the storage qualification commands.

State 1: ST1:
if ( condition1 )
    start_store;
else if ( condition2 )
    trigger value;
else if ( condition3 )
    stop_store;

Figure 9-34 shows a hypothetical scenario with three trigger conditions that happen at different times after you click Start Analysis. The trigger flow description in the example above, when applied to the scenario shown in Figure 9-34, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Figure 9-34: Capture Scenario for Storage Qualification with the State-Based Trigger Flow

In this example, the SignalTap II Logic Analyzer does not write into the acquisition buffer until sample a, when Condition 1 occurs. Once sample b is reached, the trigger value command is evaluated. The logic analyzer continues to write into the buffer to finish the acquisition. The trigger flow specifies a stop_store command at sample c, m samples after the trigger point occurs.

The logic analyzer finishes the acquisition and displays the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than m.

If the post-fill count value specified in Trigger Flow description 1 is greater than m samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again. The SignalTap II Logic Analyzer continues to evaluate the stop_store and start_store commands even after the trigger command is evaluated. If the acquisition has paused, you can click Stop Analysis to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state are updated in real-time during a data acquisition.

Figure 9-35 and Figure 9-36 show a real data acquisition of the scenario. Figure 9-35 illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, \( m = n = 10 \), and the post-fill count value \( = 5 \). Figure 9-36 illustrates a scenario where the logic analyzer pauses indefinitely even after a trigger condition occurs due to a stop_store condition. This scenario uses a sample depth of 64, with \( m = n = 10 \) and post-fill count \( = 15 \).
Figure 9-35: Storage Qualification with Post-Fill Count Value Less than $m$ (Acquisition Successfully Completes)
Figure 9-36: Storage Qualification with Post-Fill Count Value Greater than $m$ (Acquisition Indefinitely Paused)

Figure 9-37: Waveform After Forcing the Analysis to Stop
The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that are written into the acquisition buffer. The code example below shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. Figure 9-38 shows the data transaction on a continuous capture and Figure 9-40 shows the data capture with the Trigger flow description applied, in the example below.

```
State 1: ST1
start_store
if ( condition1 )
begin
  stop_store;
  goto ST2;
end

State 2: ST2
if (c1 < 3)
  increment c1; //skip three clock cycles; c1 initialized to 0
else if (c1 == 3)
begin
  start_store; //start_store necessary to enable writing to finish
  //acquisition
  trigger;
end
```

**Figure 9-38: Continuous Capture of Data Transaction for Example 2**

**Figure 9-39: Capture of Data Transaction with Trigger Flow Description Applied**

---

**Specifying the Trigger Position**

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can specify the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

- **Pre**—Saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—Saves 50% pre-trigger and 50% post-trigger data.
- **Post**—Saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).
These pre-defined ratios apply to both non-segmented buffers and segmented buffers.

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and `trigger` actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the non-segmented buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer.

\[
\text{Sample Number of Trigger Position} = (N - \text{Post-Fill Count})
\]

In this case, \( N \) is the sample depth of either the acquisition segment or non-segmented buffer.

For segmented buffers, the acquisition segments that have a post-count argument define use of the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the `Setup` tab.

**Related Information**

*State-Based Triggering* on page 9-32

---

### Creating a Power-Up Trigger

Typically, the SignalTap II Logic Analyzer is used to trigger on events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the JTAG connection for the device is available. However, there may be cases when you would like to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you arm the SignalTap II Logic Analyzer and capture data immediately after device programming.

### Enabling a Power-Up Trigger

You can add a different Power-Up Trigger to each logic analyzer instance in the SignalTap II Instance Manager pane. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click *Enable Power-Up Trigger*, or select the instance, and on the Edit menu, click *Enable Power-Up Trigger*. To disable a Power-Up Trigger, click *Disable Power-Up Trigger* in the same locations. Power-Up Trigger is shown as a child instance below the name of the selected instance with the default trigger conditions specified in the node list.
Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you can create basic and advanced trigger conditions for the trigger as you do with a Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white. Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the Instance Manager.

You cannot make changes to Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic and advanced triggers. To apply these changes to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.

**Note:** Any change made to the Power-Up Trigger conditions requires that you recompile the SignalTap II Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This enables you to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the Instance Manager and click *Duplicate Trigger*, or select the instance name or the Power-Up Trigger name and on the Edit menu, click *Duplicate Trigger*.

You can also use In-System Sources and Probes in conjunction with the SignalTap II Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain.
Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1, is evaluated, and must be TRUE before any other configured trigger conditions are evaluated. The logic analyzer supplies a signal to trigger external devices or other SignalTap II Logic Analyzer instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

You can use external triggers to perform cross-triggering on a hard processor system (HPS). Use your processor debugger to configure the HPS to obey or disregard cross-trigger request from the FPGA, and to issue or not issue cross-trigger requests to the FPGA. Use your processor debugger in combination with the SignalTap II external trigger feature to develop a dynamic combination of cross-trigger behaviors. You can use the cross-triggering feature with the ARM Development Studio 5 (DS-5) software to implement a system-level debugging solution for your Altera SoC.

Compile the Design

When you add an .stp to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection you use to control the logic analyzer. When you are debugging with a traditional external logic analyzer, you must often make changes to the signals monitored as well as the trigger conditions.

Preventing Changes Requiring Recompilation

You can configure the .stp to prevent changes that normally require recompilation. To do this, select a lock mode from above the node list in the Setup tab. To lock your configuration, choose to allow only trigger condition changes.

Timing Preservation with the SignalTap II Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successful operation of your design. The Quartus Prime Pro Edition software supports timing preservation for post-fit taps in Arria 10 designs with the Rapid Recompile feature. Rapid Recompile automatically reuses verified portions of your design during recompilations, rather than reprocessing those portions.
Use the following techniques to help maintain timing:

- Avoid adding critical path signals to your .stp.
- Minimize the number of combinational signals you add to your .stp and add registers whenever possible.
- Specify an $f_{\text{MAX}}$ constraint for each clock in your design.

Related Information

Timing Closure and Optimization documentation

Performance and Resource Considerations

There is a necessary trade-off between the runtime flexibility of the SignalTap II Logic Analyzer, the timing performance of the SignalTap II Logic Analyzer, and resource usage. The SignalTap II Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to provide maximum flexibility so you can complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The following tips provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Logic Analyzer consumes if your design is resource-constrained.

If SignalTap II logic is part of your critical path, follow these tips to speed up the performance of the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you use either advanced triggers or State-based triggering flow, disable runtime configurable parameters for a boost in $f_{\text{MAX}}$ of the SignalTap II logic. If you are using State-based triggering flow, try disabling the **Goto state destination** option and performing a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on $f_{\text{MAX}}$, as compared to the other runtime configurable options.

- **Minimize the number of signals that have Trigger Enable selected**—All signals that you add to the .stp have Trigger Enable turned on. Turn off Trigger Enable for signals that you do not plan to use as triggers.

- **Turn on Physical Synthesis for register retiming**—If you have a large number of triggering signals enabled (greater than the number of inputs that would fit in a LAB) that fan-in logic to a gate-based triggering condition, such as a basic trigger condition or a logical reduction operator in the advanced trigger tab, turn on **Perform register retiming**. This can help balance combinational logic across LABs.
If your design is resource constrained, follow these tips to reduce the amount of logic or memory used by the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in using fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Logic Analyzer by limiting the number of segments in your sampling buffer to only those required.
- **Disable the Data Enable for signals that are used for triggering only**—By default, both the **data enable** and **trigger enable** options are selected for all signals. Turning off the **data enable** option for signals used as trigger inputs only saves on memory resources used by the SignalTap II Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

**Program the Target Device or Devices**

After you compile your project, including the SignalTap II Logic Analyzer, configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, configure the device from the .stp instead of the Quartus Prime Programmer. Because you configure from the .stp, you can open more than one .stp and program multiple devices to debug multiple designs simultaneously.

The settings in an .stp must be compatible with the programming .sof used to program the device. An .stp is considered compatible with an .sof when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device is programmed. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the SignalTap II Logic Analyzer Editor.

**Note:** When the SignalTap II Logic Analyzer detects incompatibility after analysis is started, a system error message is generated containing two CRC values, the expected value and the value retrieved from the .stp instance on the device. The CRC values are calculated based on all SignalTap II settings that affect the compilation.

To ensure programming compatibility, make sure to program your device with the latest .sof created from the most recent compilation. Checking whether or not a particular .sof is compatible with the current SignalTap II configuration is achieved quickly by attaching the .sof to the SOF manager.

Before starting a debugging session, do not make any changes to the .stp settings that would require recompiling the project. You can check the SignalTap II status display at the top of the Instance Manager pane to verify whether a change you made requires recompiling the project, producing a new .sof. This feature gives you the opportunity to undo the change, so that you do not need to recompile your project. To prevent these types of changes, select **Allow trigger condition changes only** to lock the .stp. The Incremental Route lock mode, **Allow incremental route changes only**, limits changes that require an incremental route using Rapid Recompile, and not a full compile.

Although having a Quartus Prime project is not required when using an .stp, it is recommended. The project database contains information about the integrity of the current SignalTap II Logic Analyzer session. Without the project database, there is no way to verify that the current .stp matches the .sof that is downloaded to the device. If you have an .stp that does not match the .sof, incorrect data is captured in the SignalTap II Logic Analyzer.
Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, perform debugging operations in a manner similar to when you use an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the .stp file with the JTAG connection.

You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring. Figure 9-41 illustrates a flow that shows how you operate the SignalTap II Logic Analyzer. The flowchart indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 9-41: Power-Up and Runtime Trigger Events Flowchart

You can also use In-System Sources and Probes in conjunction with the SignalTap II Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected signals over the JTAG chain.
Runtime Reconfigurable Options

Certain settings in the .stp are changeable without recompiling your design when you use Runtime Trigger mode.

Table 9-10: Runtime Reconfigurable Features

<table>
<thead>
<tr>
<th>Runtime Reconfigurable Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Trigger Conditions and Basic Storage Qualifier Conditions</td>
<td>All signals that have the Trigger condition turned on can be changed to any basic trigger condition value without recompiling.</td>
</tr>
<tr>
<td>Advanced Trigger Conditions and Advanced Storage Qualifier Conditions</td>
<td>Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings are shown with a white background in the block representation. This runtime reconfigurable option is turned on in the Object Properties dialog box.</td>
</tr>
<tr>
<td>Switching between a storage-qualified and a continuous acquisition</td>
<td>Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on disable storage qualifier.</td>
</tr>
<tr>
<td>State-based trigger flow parameters</td>
<td>Table 9-5 lists Reconfigurable State-based trigger flow options.</td>
</tr>
</tbody>
</table>

Runtime Reconfigurable options can potentially save time during the debugging cycle by allowing you to cover a wider possible scenario of events without the need to recompile the design. You may experience a slight impact to the performance and logic utilization. You can turn off Runtime re-configurability for Advanced Trigger Conditions and the State-based trigger flow parameters, boosting performance and decreasing area utilization.

You can configure the .stp to prevent changes that normally require recompilation. To do this, in the Setup tab, select Allow Trigger Condition changes only above the node list.

Incremental Route lock mode, Allow incremental route changes only, limits changes which will only require an Incremental Route compilation, and not a full compile.

The example below illustrates a potential use case for Runtime Reconfigurable features. This example provides a storage qualified enabled State-based trigger flow description and shows how you can modify the size of a capture window at runtime without a recompile. This example gives you equivalent functionality to a segmented buffer with a single trigger condition where the segment sizes are runtime reconfigurable.

```
state ST1:
  if ( condition1 && (c1 <= m) )    // each "segment" triggers on condition
  begin                                     // m = number of total "segments"
    start_store;
    increment c1;
    goto ST2;
  End

  else (c1 > m )                    //This else condition handles the last
  begin                              //segment.
    start_store
```
Trigger (n-1)
end

state ST2:
if ( c2 >= n)                       //n = number of samples to capture in each
  //segment.
begin
  reset c2;
  stop_store;
  goto ST1;
end
else (c2 < n)
begin
  increment c2;
  goto ST2;
end

Note: m x n must equal the sample depth to efficiently use the space in the sample buffer.

Figure 9-42 shows a segmented buffer described by the trigger flow in example above.

During runtime, the values m and n are runtime reconfigurable. By changing the m and n values in the
preceding trigger flow description, you can dynamically adjust the segment boundaries without incurring
a recompile.

Figure 9-42: Segmented Buffer Created with Storage Qualifier and State-Based Trigger

Total sample depth is fixed, where m x n must equal sample depth.

You can add states into the trigger flow description and selectively mask out specific states and enable
other ones at runtime with status flags.

The example below shows a modified description of the example above with an additional state inserted.
You use this extra state to specify a different trigger condition that does not use the storage qualifier
feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

state ST1 :
if (condition2  && f1)                       //additional state added for a non-
  //acquisition   Set f1 to enable state
begin
  start_store;
  trigger
end
else if (! f1)
goto ST2;
state ST2:
if ( (condition1 && (c1 <= m)   && f2)   //f2  status flag used to mask state. Set
  //to enable.
begin
  start_store;
  increment c1;
  goto ST3;
end
else (c1 > m )
SignalTap II Status Messages

Table 9-11 describes the text messages that might appear in the SignalTap II Status Indicator in the Instance Manager pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 9-11: Text Messages in the SignalTap II Status Indicator

<table>
<thead>
<tr>
<th>Message</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not running</td>
<td>The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.</td>
</tr>
<tr>
<td>(Power-Up Trigger) Waiting for clock (1)</td>
<td>The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.</td>
</tr>
<tr>
<td>Acquiring (Power-Up) pre-trigger data (1)</td>
<td>The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.</td>
</tr>
<tr>
<td>Trigger In conditions met</td>
<td>Trigger In condition has occurred. The SignalTap II Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.</td>
</tr>
<tr>
<td>Waiting for (Power-up) trigger (1)</td>
<td>The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.</td>
</tr>
<tr>
<td>Trigger level &lt;x&gt; met</td>
<td>The condition of trigger condition x has occurred. The SignalTap II Logic Analyzer is waiting for the condition specified in condition x + 1 to occur.</td>
</tr>
<tr>
<td>Acquiring (power-up) post-trigger data (1)</td>
<td>The entire trigger event has occurred. The SignalTap II Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.</td>
</tr>
<tr>
<td>Offload acquired (Power-Up) data (1)</td>
<td>Data is being transmitted to the Quartus Prime software through the JTAG chain.</td>
</tr>
<tr>
<td>Ready to acquire</td>
<td>The SignalTap II Logic Analyzer is waiting for you to initialize the analyzer.</td>
</tr>
<tr>
<td>Message</td>
<td>Message Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td><strong>Note to Table 9-11:</strong></td>
<td>1. This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses is added.</td>
</tr>
</tbody>
</table>

**Note:** In segmented acquisition mode, pre-trigger and post-trigger do not apply.

**View, Analyze, and Use Captured Data**

Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design.

When in the Data view, you can use the drag-to-zoom feature by left-clicking to isolate the data of interest.

**Capturing Data Using Segmented Buffers**

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently. Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the SignalTap II Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. The trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, is defined by either the Sequential trigger flow control or the Custom State-based trigger flow control. **Figure 9-43** shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

**Figure 9-43: Segmented Acquisition Buffer**

The SignalTap II Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. **Figure 9-43** illustrates the method in which data is captured. The Trigger markers in **Figure 9-43**—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. If you use a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the SignalTap II Logic Analyzer starts evaluating Trigger 2 immediately. Data Acquisition for Segment 2 buffer starts when either Segment Buffer 1 finishes its post-fill count, or when Trigger 2 evaluates as **TRUE**, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This allows the SignalTap II Logic Analyzer to accurately...
capture all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

**Figure 9-44** shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **Don’t Care**. Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all segments is specified as pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the SignalTap II Logic Analyzer allocated to the buffer.

**Figure 9-44: Segmented Capture with Preemption of Acquisition Segments**

A segmented acquisition buffer using the sequential trigger flow with a trigger condition specified as Don’t Care. All segments, with the exception of the last segment, capture only one sample because the next trigger condition preempts the current buffer from filling to completion.

For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. For maximum flexibility on how the trigger position is defined, use the custom state-based trigger flow. By adjusting the trigger position specific to your debugging requirements, you can help maximize the use of the allocated buffer space.

**Differences in Pre-fill Write Behavior Between Different Acquisition Modes**

The SignalTap II Logic Analyzer uses one of the following three modes when writing into the acquisition memory:

- **Non-segmented buffer**
- **Non-segmented buffer with a storage qualifier**
- **Segmented buffer**

There are subtle differences in the amount of data captured immediately after running the SignalTap II Logic Analyzer and before any trigger conditions occur. A non-segmented buffer, running in continuous mode, completely fills the buffer with sampled data before evaluating any trigger conditions. Thus, a non-segmented capture without any storage qualification enabled always shows a waveform with a full buffer’s worth of data captured.

Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, SignalTap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection and prevents the buffer from being dumped during the first acquisition prior to a trigger condition when you perform a **Stop Analysis**.

For segmented buffers and non-segmented buffers using any storage qualification mode, the SignalTap II Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. The logic analyzer evaluates each trigger condition before acquiring a full buffer’s worth of samples. This evaluation is especially important when using any storage qualification on the data set. The
logic analyzer may miss a trigger condition if it waits until a full buffer’s worth of data is captured before evaluating any trigger conditions.

If the trigger event occurs on any data sample before the specified amount of pre-trigger data has occurred, then the SignalTap II Logic Analyzer triggers and begins filling memory with post-trigger data, regardless of the amount of pre-trigger data you specify. For example, if you set the trigger position to 50% and set the logic analyzer to trigger on a processor reset, start the logic analyzer, and then power on your target system, the logic analyzer triggers. However, the logic analyzer memory is filled only with post-trigger data, and not any pre-trigger data, because the trigger event, which has higher precedence than the capture of pre-trigger data, occurred before the pre-trigger condition was satisfied.

Figure 9-45 and Figure 9-46 show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The logic analyzer for the waveforms below is configured with a sample depth of 64 bits, with a trigger position specified as Post trigger position.

Figure 9-45: SignalTap II Logic Analyzer Continuous Data Capture

![Figure 9-45: SignalTap II Logic Analyzer Continuous Data Capture](image)

Note to Figure 9-45:
1. Continuous capture mode with post-trigger position.
2. Capture of a recurring pattern using a non-segmented buffer in continuous mode. The SignalTap II Logic Analyzer is configured with a basic trigger condition (shown in the figure as "Trig1") with a sample depth of 64 bits.

Notice in Figure 9-45 that Trig1 occurs several times in the data buffer before the SignalTap II Logic Analyzer actually triggers. A full buffer’s worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 9-46: SignalTap II Logic Analyzer Conditional Data Capture

![Figure 9-46: SignalTap II Logic Analyzer Conditional Data Capture](image)

Note to Figure 9-46:
1. Conditional capture, storage always enabled, post-fill count.
2. SignalTap II Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The logic analyzer is configured with a basic trigger condition (shown in the figure
as "Trig1"), with a sample depth of 64 bits. The “Trigger in” condition is specified as "Don't care", which means that every sample is captured.

Notice in Figure 9-46 that the logic analyzer triggers immediately. As in Figure 9-45, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the Setup or Data tab of an .stp and click Mnemonic Table Setup. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, assign the table to a group of signals. To assign a mnemonic table, right-click on the group, click Bus Display Format and select the desired mnemonic table.

You use the labels you create in a table in different ways on the Setup and Data tabs. On the Setup tab, you can create basic triggers with meaningful names by right-clicking an entry in the Trigger Conditions column and selecting a label from the table you assigned to the signal group. On the Data tab, if any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to an .stp, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click on the name of the signal or signal group. On the Bus Display Format shortcut menu, then click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in helps you to monitor signal activity for your design as the code is executed. If you set up the logic analyzer to trigger on a function name in your Nios II code based on data from an .elf, you can see the function name in the Instance Address signal group at the trigger sample, along with the corresponding disassembled code in the Disassembly signal group, as shown in Figure 9-47. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 9-47: Data Tab when the Nios II Plug-In is Used

Locating a Node in the Design

When you find the source of an error in your design using the SignalTap II Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Quartus Prime software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your
design to correct the flaw. To locate a signal from the SignalTap II Logic Analyzer in one of the Quartus Prime software tools or your design files, right-click on the signal in the .stp, and click **Locate in** *<tool name>*.

You can locate a signal from the node list with the following tools:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

### Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The SignalTap II Logic Analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This allows you to review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. To open the **Data Log** pane, on the View menu, select **Data Log**. To turn on data logging, turn on **Enable data log** in the **Data Log** *(Figure 9-19)*. To recall and activate a data log for a given trigger set, double-click the name of the data log in the list. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.

### Related Information

**Managing Multiple SignalTap II Files and Configurations** on page 9-21

You can use the Data Log feature for organizing different sets of trigger conditions and different sets of signal configurations.

### Exporting Captured Data to Other File Formats

You can export captured data to the following file formats, for use with other EDA simulation tools:

- Comma Separated Values File (**.csv**)
- Table File (**.tbl**)
- Value Change Dump File (**.vcd**)
- Vector Waveform File (**.vwf**)
- Graphics format files (**.jpg, .bmp**)

To export the captured data from SignalTap II Logic Analyzer, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.
Creating a SignalTap II List File

Captured data can also be viewed in an .stp list file. An .stp list file is a text file that lists all the data captured by the logic analyzer for a trigger event. Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the same time period of the trigger event. To create an .stp list file in the Quartus Prime software, on the File menu, select Create/Update and click Create SignalTap II List File.

Other Features

The SignalTap II Logic Analyzer has other features that do not necessarily belong to a particular task in the task flow.

Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function alt_signaltap_run, built into the Quartus Prime software, to acquire data from the SignalTap II Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using SignalTap II in the Quartus Prime software environment.

Note: The SignalTap II MATLAB MEX function is available in the Windows version and Linux version of the Quartus Prime software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus Prime software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus Prime software, create an .stp file.
2. In the node list in the Data tab of the SignalTap II Logic Analyzer Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix. Each column of the imported matrix represents a single SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the Data tab.

Note: Signal groups acquired from the SignalTap II Logic Analyzer and transferred into the MATLAB MEX function are limited to a width of 32 signals. If you want to use the MATLAB MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the 32-signal limit.

3. Save the .stp and compile your design. Program your device and run the SignalTap II Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Quartus Prime binary directory to your path with the following command:

```matlab
addpath <Quartus install directory>\win
```

You can view the help file for the MEX function by entering the following command in MATLAB without any operators:

```
altsignaltap_run
```
Use the MATLAB MEX function to open the JTAG connection to the device and run the SignalTap II Logic Analyzer to acquire data. When you finish acquiring data, close the JTAG connection.

To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called stp, use the following command:

```matlab
stp = alt_signaltap_run('"<stp filename>"', {'signed' | 'unsigned'}, {'instance names'}, {'<signalset name>'}, {'<trigger name>'});
```

When capturing data you must assign a filename, for example, `<stp filename>` as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in Table 9-12.

**Table 9-12: SignalTap II MATLAB MEX Function Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>'signed'</td>
<td>The <code>signed</code> option turns signal group data into 32-bit two’s-complement signed integers. The MSB of the group as defined in the SignalTap II Data tab is the sign bit. The <code>unsigned</code> option keeps the data as an unsigned integer. The default is <code>signed</code>.</td>
</tr>
<tr>
<td>unsigned</td>
<td>'unsigned'</td>
<td></td>
</tr>
<tr>
<td><code>&lt;instance name&gt;</code></td>
<td>'auto_signaltap_0'</td>
<td>Specify a SignalTap II instance if more than one instance is defined. The default is the first instance in the .stp, <code>auto_signaltap_0</code>.</td>
</tr>
<tr>
<td><code>&lt;signal set name&gt;</code></td>
<td>'my_signalset'</td>
<td>Specify the signal set and trigger from the SignalTap II data log if multiple configurations are present in the .stp. The default is the active signal set and trigger in the file.</td>
</tr>
<tr>
<td><code>&lt;trigger name&gt;</code></td>
<td>'my_trigger'</td>
<td></td>
</tr>
</tbody>
</table>

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. To enable or disable verbose mode, use the following commands:

```matlab
alt_signaltap_run('VERBOSE_ON');
alt_signaltap_run('VERBOSE_OFF');
```

When you finish acquiring data, close the JTAG connection with the following command:

```matlab
alt_signaltap_run('END_CONNECTION');
```

For more information about the use of MATLAB MEX functions in MATLAB, refer to the MATLAB Help.

**Using SignalTap II in a Lab Environment**

You can install a stand-alone version of the SignalTap II Logic Analyzer. This version is particularly useful in a lab environment in which you do not have a workstation that meets the requirements for a complete Quartus Prime installation, or if you do not have a license for a full installation of the Quartus Prime software. The standalone version of the SignalTap II Logic Analyzer is included with and requires the Quartus Prime stand-alone Programmer which is available from the Downloads page of the Altera website.
Remote Debugging Using the SignalTap II Logic Analyzer

Debugging Using a Local PC and an Altera SoC
You can use the System Console with SignalTap II Logic Analyzer to remote debug your Altera SoC. This method requires one local PC, an existing TCP/IP connection, a programming device at the remote location, and an Altera SoC.

Related Information
Remote Hardware Debugging over TCP/IP for Altera SoC application note

Debugging Using a Local PC and a Remote PC
You can use the SignalTap II Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Quartus Prime software installed on the local PC
- Stand-alone SignalTap II Logic Analyzer or the full version of the Quartus Prime software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

Equipment Setup
On the PC in the remote location, install the standalone version of the SignalTap II Logic Analyzer, included in the Quartus Prime standalone Programmer, or the full version of the Quartus Prime software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

On the local PC, install the full version of the Quartus Prime software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

Using the SignalTap II Logic Analyzer in Devices with Configuration Bitstream

Security
Certain device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the SignalTap II Logic Analyzer to analyze functional data within the FPGA. However, note that JTAG configuration is not possible after the security key has been programmed into the device.

Altera recommends that you use an unencrypted bitstream during the prototype and debugging phases of the design. Using an unencrypted bitstream allows you to generate new programming files and reconfigure the device over the JTAG connection during the debugging cycle.

If you must use the SignalTap II Logic Analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the SignalTap II Logic Analyzer. After the FPGA is configured with a SignalTap II Logic Analyzer instance in the design, when you open the SignalTap II Logic Analyzer in the Quartus Prime software, you then scan the chain and are ready to acquire data with the JTAG connection.
Backward Compatibility with Previous Versions of Quartus Prime Software

You can open an .stp created in a previous version in a current version of the Quartus Prime software. However, opening an .stp modifies it so that it cannot be opened in a previous version of the Quartus Prime software.

If you have a Quartus Prime project file from a previous version of the software, you may have to update the .stp configuration file to recompile the project. You can update the configuration file by opening the SignalTap II Logic Analyzer. If you need to update your configuration, a prompt appears asking if you would like to update the .stp to match the current version of the Quartus Prime software.

SignalTap II Command-Line Options

To compile your design with the SignalTap II Logic Analyzer using the command prompt, use the quartus_stp command. Table 9-13 shows the options that help you use the quartus_stp executable.

Table 9-13: SignalTap II Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stp_file</td>
<td>quartus_stp --stp_file &lt;stp_filename&gt;</td>
<td>Assigns the specified .stp to the USE_SIGNALTAP_FILE in the .qsf.</td>
</tr>
<tr>
<td>enable</td>
<td>quartus_stp --enable</td>
<td>Creates assignments to the specified .stp in the .qsf and changes ENABLE_SIGNALTAP to ON. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no .stp is specified in the .qsf, the --stp_file option must be used. If the --enable option is omitted, the current value of ENABLE_SIGNALTAP in the .qsf is used.</td>
</tr>
<tr>
<td>disable</td>
<td>quartus_stp --disable</td>
<td>Removes the .stp reference from the .qsf and changes ENABLE_SIGNALTAP to OFF. The SignalTap II Logic Analyzer is removed from the design database the next time you compile your design. If the --disable option is omitted, the current value of ENABLE_SIGNALTAP in the .qsf is used.</td>
</tr>
</tbody>
</table>

The first example illustrates how to compile a design with the SignalTap II Logic Analyzer at the command line.

quartus_stp filtref --stp_file stp1.stp --enable
quartus_map filtref --source=filtref.bdf --family=CYCLONE
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns
quartus_asm filtref

The quartus_stp --stp_file stp1.stp --enable command creates the QSF variable and instructs the Quartus Prime software to compile the .stp file with your design. The --enable option must be applied for the SignalTap II Logic Analyzer to compile properly into your design.
The example below shows how to create a new .stp after building the SignalTap II Logic Analyzer instance with the IP Catalog.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp
```

**Related Information**

- **Command-Line Scripting documentation**
  Information about the other command line executables and options

### SignalTap II Tcl Commands

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Quartus Prime GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the Quartus Prime software. They must be executed from the command line with the `quartus_stp` executable.

To execute a Tcl file that has SignalTap II Logic Analyzer Tcl commands, use the following command:

```
quartus_stp -t <Tcl file>
```

The example is an excerpt from a script you can use to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

```
# opens signaltap session
open_session -name stp1.stp
# start acquisition of instance auto_signaltap_0 and
# auto_signaltap_1 at the same time
# calling run_multiple_end will start all instances
# run after run_multiple_start call
run_multiple_start
run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger \ trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger \ trigger_1 -data_log log_1 -timeout 5
run_multiple_end
# close signaltap session
close_session
```

When the script is completed, open the .stp that you used to capture data to examine the contents of the Data Log.

**Related Information**

- `::quartus::stp online help`
  Information about Tcl commands that you can use with the SignalTap II Logic Analyzer Tcl package

### Design Example: Using SignalTap II Logic Analyzers

The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button. After you press a button, the processor initiates a DMA transfer, which you analyze using the SignalTap II Logic Analyzer.

**Related Information**

- AN 446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer application note
Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the SignalTap II Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode All states in one window.

Related Information
On-chip Debugging Design Examples website

Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. The example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The acquisition stops after all four segments are filled once.

```
if (c1 == 3 && condition1)
  trigger 30;
else if ( condition1 )
begin
  segment_trigger 30;
  increment c1;
end
```

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is displayed on the Data tab as the last sample number in the affected segment. The trigger position in the affected segment is then defined by $N - \text{post count fill}$, where $N$ is the number of samples per segment. Figure 9-48 illustrates the triggering position.
Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. The example shows such a sample flow. This example uses three basic triggering conditions configured in the SignalTap II Setup tab.

This example triggers the acquisition buffer when condition1 occurs after condition3 and occurs ten times prior to condition3. If condition3 occurs prior to ten repetitions of condition1, the state machine transitions to a permanent wait state.

```
state ST1:
  if ( condition2 )
  begin
    reset c1;
    goto ST2;
  end
State ST2 :
  if ( condition1 )
    increment c1;
  else if (condition3 && c1 < 10)
    goto ST3;
  else if ( condition3 && c1 >= 10)
    trigger;
ST3:
  goto ST3;
```

SignalTap II Scripting Support

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

```
quartus_sh --qhelp
```

Related Information

- Tcl Scripting documentation
- Quartus Prime Tcl Scripting online help

### Document Revision History

#### Table 9-14: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>• Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated content to reflect SignalTap support in Quartus Prime Pro Edition</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Added content for Floating Point Display Format in table: SignalTap II Logic Analyzer Features and Benefits.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>• Added MAX 10 as supported device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Full Incremental Compilation setting and Post-Fit (Strict) netlist type setting information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed outdated GUI images from &quot;Using Incremental Compilation with the SignalTap II Logic Analyzer&quot; section.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• DITA conversion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replaced MegaWizard Plug-In Manager and Megafunctio opponent content with IP Catalog and parameter editor content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added flows for custom trigger HDL object, Incremental Route with Rapid Recompile, and nested groups with Basic OR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GUI changes: toolbar, drag to zoom, disable/enable instance, trigger log time-stamping.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy material. Added section on using cross-triggering with DS-5 tool and added link to white paper 01198. Added section on remote debugging an Altera SoC and added link to application note 693. Updated support for MEX function.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes Made</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| May 2013     | 13.0.0  | • Added recommendation to use the state-based flow for segmented buffers with separate trigger conditions, information about Basic OR trigger condition, and hard processor system (HPS) external triggers.  
| June 2012    | 12.0.0  | Updated Figure 13–5 on page 13–16 and “Adding Signals to the SignalTap II File” on page 13–10.                                           |
| November 2011| 11.0.1  | Template update.  

Minor editorial updates.                                                                                                      |
| May 2011     | 11.0.0  | Updated the requirement for the standalone SignalTap II software.                                                                       |
| December 2010| 10.0.1  | Changed to new document template.                                                                                                       |
| July 2010    | 10.0.0  | • Add new acquisition buffer content to the “View, Analyze, and Use Captured Data” section.                                             
• Added script sample for generating hexadecimal CRC values in programmed devices.                                             
• Created cross references to Quartus Prime Help for duplicated procedural content.                                             |
| November 2009| 9.1.0   | No change to content.                                                                                                                   |
| March 2009   | 9.0.0   | • Updated Table 13–1  
• Updated “Using Incremental Compilation with the SignalTap II Logic Analyzer” on page 13–45  
• Added new Figure 13–33  
• Made minor editorial updates                                                                                                      |
| November 2008| 8.1.0   | Updated for the Quartus Prime software version 8.1 release:  
• Added new section “Using the Storage Qualifier Feature” on page 14–25  
• Added description of `start_store` and `stop_store` commands in section “Trigger Condition Flow Control” on page 14–36  
• Added new section “Runtime Reconfigurable Options” on page 14–63                                                                 |

**Document Revision History**

Altera Corporation  
Design Debugging Using the SignalTap II Logic Analyzer  
Send Feedback
May 2008  8.0.0  Updated for the Quartus Prime software version 8.0:
• Added “Debugging Finite State machines” on page 14-24
• Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab
• Added “Capturing Data Using Segmented Buffers” on page 14–16
• Added hyperlinks to referenced documents throughout the chapter
• Minor editorial updates

Related Information
Quartus Handbook Archive
For previous versions of the Quartus Prime Handbook, refer to the Quartus Handbook Archive.
You can detect and debug single event upset (SEU) using the Fault Injection Debugger in the Quartus® Prime software. Use the debugger with the Altera Fault Injection IP core to inject errors into the configuration RAM (CRAM) of an FPGA device.

The injected error simulate the soft errors that can occur during normal operation due to SEUs. Because SEUs are rare events, and therefore difficult to test, you can use the Fault Injection Debugger to induce intentional errors in the FPGA to test the system's response to these errors.

The Fault Injection Debugger is available for Stratix V family devices. For assistance with support for Arria V or Cyclone V family devices, file a service request using mySupport.

The Fault Injection Debugger provides the following benefits:

- Allows you to evaluate system response for mitigating single event functional interrupts (SEFI).
- Allows you to perform SEFI characterization, eliminating the need for entire system beam testing. Instead, you can limit the beam testing to failures in time (FIT)/Mb measurement at the device level.
- Scale FIT rates according to the SEFI characterization that is relevant to your design architecture. You can randomly distribute fault injections throughout the entire device, or constrain them to specific functional areas to speed up testing.
- Optimize your design to reduce SEU-caused disruption.

Related Information

Altera Website: Single Event Upsets

Single Event Upset Mitigation

Integrated circuits and programmable logic devices such as FPGAs are susceptible to SEUs. SEUs are random, nondestructive events, caused by two major sources: alpha particles and neutrons from cosmic rays. Radiation can cause either the logic register, embedded memory bit, or a configuration RAM (CRAM) bit to flip its state, thus leading to unexpected device operation.

Arria V, Cyclone V, Stratix V and newer devices have the following CRAM capabilities:

- Error Detection Cyclic Redundance Checking (EDCRC)
- Automatic correction of an upset CRAM (scrubbing)
- Ability to create an upset CRAM condition (fault injection)
For more information about SEU mitigation in Altera devices, refer to the SEU Mitigation chapter in the respective device handbook.

Related Information
Altera Website: Single Event Upsets

Hardware and Software Requirements

The following hardware and software is required to use the Fault Injection Debugger:

- Quartus Prime software version 14.0 or later.
- FEATURE line in your Altera license that enables the Fault Injection IP core. For more information, contact your local Altera sales representative.
- Download cable (USB-Blaster, USB-Blaster II, EthernetBlaster, or EthernetBlaster II cable).
- Altera development kit or user designed board with a JTAG connection to the device under test.
- (Optional) FEATURE line in your Altera license that enables the Advanced SEU Detection IP core.

Related Information
Altera Website: Contact Altera

Using the Fault Injection Debugger and Fault Injection IP Core

The Fault Injection Debugger works together with the Fault Injection IP core. First, you instantiate the IP core in your design, compile, and download the resulting configuration file into your device. Then, you run the Fault Injection Debugger from within the Quartus Prime software or from the command line to simulate soft errors.

The Fault Injection Debugger communicates with the Fault Injection IP core via the JTAG interface. You perform debugging using the Fault Injection Debugger in the Quartus Prime software or using the command-line interface.

- The Fault Injection Debugger allows you to operate fault injection experiments interactively or by batch commands, and allows you to specify the logical areas in your design for fault injections.
- The command-line interface is useful for running the debugger via a script.

The Fault Injection IP accepts commands from the JTAG interface and reports status back through the JTAG interface.

Note: The Fault Injection IP core is implemented in soft logic in your device; therefore, you must account for this logic usage in your design. One methodology is to characterize your design’s response to SEU in the lab and then omit the IP core from your final deployed design.

You use the Fault Injection IP core with the following IP cores:

- The Error Message Register (EMR) Unloader IP core, which reads and stores data from the hardened error detection circuitry in Altera devices.
- (Optional) The Advanced SEU Detection (ASD) IP core, which compares single-bit error locations to a sensitivity map during device operation to determine whether a soft error affects it.
**Figure 10-1: Fault Injection Debugger Overview Block Diagram**

![Fault Injection Debugger Overview Block Diagram](image)

**Notes:**
1. The fault Injection IP flips the bits of the targeted logic.
2. The Fault Injection Debugger and Advanced SEU Detection IP use the same EMR Unloader instance.
3. The Advanced SEU Detection IP core is optional.

**Related Information**
- Download Center
- AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices
- Understanding Single Event Functional Interrupts in FPGA Designs White Paper
- Altera Fault Injection IP Core User Guide
- Altera Error Message Unloader IP Core User Guide
- Altera Advanced SEU Detection (ALTERA_ADV_SEU_DETECTION) IP Core User Guide

**Instantiating the Fault Injection IP Core**

The Fault Injection IP core does not require you to set any parameters. To use the IP core, create a new IP instance, include it in your Qsys system, and connect the signals as appropriate.

**Note:** You must use the Fault Injection IP core with the Error Message Register (EMR) Unloader IP core.

The Fault Injection and the EMR Unloader IP cores are available in Qsys and the IP Catalog. Optionally, you can instantiate them directly into your RTL design, using Verilog HDL, SystemVerilog, or VHDL.

**Related Information**
- Altera Fault Injection IP Core User Guide

**Using the EMR Unloader IP Core**

The EMR Unloader IP core provides an interface to the EMR, which is updated continuously by the device’s EDCRC that checks the device’s CRAM bits CRC for soft errors.
Using the Advanced SEU Detection IP Core

Use the Advanced SEU Detection (ASD) IP core when SEU tolerance is a design concern.
You must use the EMR Unloader IP core with the ASD IP core. Therefore, if you use the ASD IP and the Fault Injection IP in the same design, they must share the EMR Unloader output via an Avalon-ST splitter component. The following figure shows a Qsys system in which an Avalon-ST splitter distributes the EMR contents to the ASD and Fault Injection IP cores.

**Figure 10-4: Using the ASD and Fault Injection IP in the Same Qsys System**

Related Information

- [Altera Advanced SEU Detection (ALTERA_ADV_SEU_DETECTION) IP Core User Guide](#)

**Defining Fault Injection Areas**

You can define specific regions of the FPGA for fault injection using a Sensitivity Map Header (`.smh`) file. The SMH file stores the coordinates of the device CRAM bits, their assigned region (ASD Region), and criticality. During the design process you use hierarchy tagging to create the region. Then, during compilation, the Quartus Prime Assembler generates the SMH file. The Fault Injection Debugger limits error injections to specific device regions you define in the SMH file.
Performing Hierarchy Tagging

You define the FPGA regions for testing by assigning an ASD Region to the location. You can specify an ASD Region value for any portion of your design hierarchy using the Design Partitions Window.

1. Choose Assignments > Design Partitions Window.
2. Right-click anywhere in the header row and turn on ASD Region to display the ASD Region column (if it is not already displayed).
3. Enter a value from 0 to 16 for any partition to assign it to a specific ASD Region.
   - ASD region 0 is reserved for unused portions of the device. You can assign a partition to this region to specify it as non-critical.
   - ASD region 1 is the default region. All used portions of the device are assigned to this region unless you explicitly change the ASD Region assignment.

About SMH Files

The SMH file contains the following information:

- If you are not using hierarchy tagging (i.e., the design has no explicit ASD Region assignments in the design hierarchy), the SMH file lists every CRAM bit and indicates whether it is sensitive for the design.
- If you have performed hierarchy tagging and changed default ASD Region assignments, the SMH file lists every CRAM bit and its assigned ASD region.

The Fault Injection Debugger can limit injections to one or more specified regions.

Note: To direct the Assembler to generate an SMH file:

- Choose Assignments > Device > Device and Pin Options > Error Detection CRC.
- Turn on the Generate SEU sensitivity map file (.smh) option.

Using the Fault Injection Debugger

To use the Fault Injection Debugger, you connect from the tool to the device via the JTAG interface. Then, configure the device and perform fault injection.

To launch the Fault Injection Debugger, choose Tools > Fault Injection Debugger in the Quartus Prime software.

Note: Configuring or programming the device is similar to the procedure used for the Programmer or SignalTap II Logic Analyzer.
To configure your JTAG chain:

1. Click **Hardware Setup**. The tool displays the programming hardware connected to your computer.
2. Select the programming hardware you wish to use.
3. Click **Close**.
4. Click **Auto Detect**, which populates the device chain with the programmable devices found in the JTAG chain.

**Related Information**

**Targeted Fault Injection Feature** on page 10-13

**Configuring Your Device and the Fault Injection Debugger**

The Fault Injection Debugger uses a `.sof` and (optionally) a Sensitivity Map Header (`.smh`) file.

The Software Object File (`.sof`) configures the FPGA. The `.smh` file defines the sensitivity of the CRAM bits in the device. If you do not provide an `.smh` file, the Fault Injection Debugger injects faults randomly throughout the CRAM bits.
To specify a .sof:

1. Select the FPGA you wish to configure in the Device chain box.
2. Click Select File.
3. Navigate to the .sof and click OK. The Fault Injection Debugger reads the .sof.
4. (Optional) Select the SMH file.
   a. Right-click the device in the Device chain box and then click Select SMH File.
   b. Select your SMH file.
   c. Click OK.
5. Turn on Program/Configure.
6. Click Start.

The Fault Injection Debugger configures the device using the .sof.

Figure 10-6: Context Menu for Selecting the SMH File

Constraining Regions for Fault Injection

After loading an SMH file, you can direct the Fault Injection Debugger to operate on only specific ASD regions.
To specify the ASD region(s) in which to inject faults:

1. Right-click on the FPGA in the **Device chain** box and click Show Device Sensitivity Map.
2. Select the ASD region(s) for fault injection.

**Figure 10-7: Sensitivity Map Viewer**

---

**Specifying Error Types**

You can specify various types of errors for injection.

- Single errors (SE)
- Double-adjacent errors (DAE)
- Uncorrectable multi-bit errors (EMBE)

Altera devices can self-correct single and double-adjacent errors if the scrubbing feature is enabled. Altera devices cannot correct multi-bit errors. Refer to the chapter on mitigating SEUs for more information about debugging these errors.

You can specify the mixture of faults to inject and the injection time interval. To specify the injection time interval:
1. In the Fault Injection Debugger, choose **Tools > Options**.
2. Drag the red controller to the mix of errors. Alternatively, you can specify the mix numerically.
3. Specify the **Injection interval time**.
4. Click **OK**.

**Figure 10-8: Specifying the Mixture of SEU Fault Types**

**Related Information**

**Mitigating Single Event Upsets**

**Injecting Errors**

You can inject errors in several modes:

- Inject one error on command
- Inject multiple errors on command
- Inject errors until commanded to stop

To inject these faults:

1. Turn on the **Inject Fault** option.
2. Choose whether you want to run error injection for a number of iterations or until stopped:
   - If you choose to run until stopped, the Fault Injection Debugger injects errors at the interval specified in the **Tools > Options** dialog box.
   - If you want to run error injection for a specific number of iterations, enter the number.
3. Click **Start**.

   **Note:** The Fault Injection Debugger runs for the specified number of iterations or until stopped.

The Quartus Prime Messages window shows messages about the errors that are injected. For additional information on the injected faults, click **Read EMR**. The Fault Injection Debugger reads the device’s EMR and displays the contents in the Messages window.
Recording Errors

You can record the location of any injected fault by noting the parameters reported in the Quartus Prime Messages window.

If, for example, an injected fault results in behavior you would like to replay, you can target that location for injection. You perform targeted injection using the Fault Injection Debugger command line interface.

Clearing Injected Errors

To restore the normal function of the FPGA, click Scrub. When you scrub an error, the device’s EDCRC functions are used to correct the errors. The scrub mechanism is similar to that used during device operation.

Command-Line Interface

You can run the Fault Injection Debugger at the command line with the `quartus_fid` executable, which is useful if you want to perform fault injection from a script.

Table 10-1: Command line Arguments for Fault Injection

<table>
<thead>
<tr>
<th>Short Argument</th>
<th>Long Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>cable</td>
<td>Specify programming hardware or cable. (Required)</td>
</tr>
<tr>
<td>i</td>
<td>index</td>
<td>Specify the active device to inject fault. (Required)</td>
</tr>
<tr>
<td>n</td>
<td>number</td>
<td>Specify the number of errors to inject. The default value is 1. (Optional)</td>
</tr>
<tr>
<td>t</td>
<td>time</td>
<td>Interval time between injections. (Optional)</td>
</tr>
</tbody>
</table>

*Note: Use `quartus_fid --help` to view all available options.*

The following code provides examples using the Fault Injection Debugger command-line interface.

```bash
# Find out which USB cables are available for this instance
```
# The result shows that one cable is available, named "USB-Blaster"

```
$ quartus_fid --list
...
Info: Command: quartus_fid --list
1) USB-Blaster on sj-sng-z4 [USB-0]
Info: Quartus Prime 64-Bit Fault Injection Debugger was successful. 0 errors, 0 warning
```

# Find which devices are available on USB-Blaster cable
# The result shows two devices: a Stratix V A7, and a MAX V CPLD.

```
$ quartus_fid --cable USB-Blaster -a
Info: Command: quartus_fid --cable=USB-Blaster -a
Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
1) USB-Blaster on sj-sng-z4 [USB-0]
   029030DD  5SGXEA7H(1|2|3)/5SGXEA7K1/..
   020A40DD  5M2210Z/EPM2210
Info: Quartus Prime 64-Bit Fault Injection Debugger was successful. 0 errors, 0 warnings
```

# Program the Stratix V device
# The --index option specifies operations performed on a connected device.
# "@1=svgx.sof" associates a .sof file with the device
# ":p" means program the device

```
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#p"
...
Info (209016): Configuring device index 1
Info (209017): Device 1 contains JTAG ID code 0x029030DD
Info (209007): Configuration succeeded -- 1 device(s) configured
Info (209011): Successfully performed operation(s)
Info (208551): Program signature into device 1.
Info: Quartus Prime 64-Bit Fault Injection Debugger was successful. 0 errors, 0 warnings
```

# Inject a fault into the device.
# The #i operator indicates to inject faults
# -n 3 indicates to inject 3 faults

```
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#! -n 3
Info: Command: quartus_fid --cable=USB-Blaster --index=1=svgx.sof#! -n 3
Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
Info (208521): Injects 3 error(s) into device(s)
Info: Quartus Prime 64-Bit Fault Injection Debugger was successful. 0 errors, 0 warnings
```

# Interactive Mode.
# Using the #i operation with -n 0 puts the debugger into interactive mode.
# Note that 3 faults were injected in the previous session;
# "E" reads the faults currently in the EMR Unloader IP core.

```
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#i" -n 0
Info: Command: quartus_fid --cable=USB-Blaster --index=1=svgx.sof#i -n 0
Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
Enter :
   'F' to inject fault
   'E' to read EMR
   'S' to scrub error(s)
   'Q' to quit
```

Altera Corporation  Debugging Single Event Upset Using the Fault Injection Debugger
Targeted Fault Injection Feature

The Fault Injection Debugger injects faults into the FPGA randomly. However, the Targeted Fault Injection feature allows you to inject faults into targeted locations in the CRAM. This operation may be useful, for example, if you noted an SEU event and want to test the FPGA or system response to the same event after modifying a recovery strategy.

Note: The Targeted Fault Injection feature is available only from the command line interface.

You can specify that errors are injected from the command line or in prompt mode.

Related Information

AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices

Specifying an Error List From the Command Line

The Targeted Fault Injection feature allows you to specify an error list from the command line, as shown in the following example:

c:\Users\sng> quartus_fid -c 1 -i "@1=svgx.sof#i" -n 2 -user="@1= 0x2274 0x05EF 0x2264 0x0500"

Where:

c 1 indicates that the fpga is controlled by the first cable on your computer.

i "@1=svgx.sof#i" indicates that the first device in the chain is loaded with the object file svgx.sof and will be injected with faults.

n 2 indicates that two faults will be injected.

user="@1= 0x2274 0x05EF 0x2264 0x0500" is a user-specified list of faults to be injected. In this example, device 1 has two faults: at frame 0x2274, bit 0x05EF and at frame 0x2264, bit 0x0500.
Specifying an Error List From Prompt Mode

You can operate the Targeted Fault Injection feature interactively by specifying the number of faults to be 0 (-n 0). The Fault Injection Debugger presents prompt mode commands and their descriptions.

<table>
<thead>
<tr>
<th>Prompt Mode Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Inject a fault</td>
</tr>
<tr>
<td>E</td>
<td>Read the EMR</td>
</tr>
<tr>
<td>S</td>
<td>Scrub errors</td>
</tr>
<tr>
<td>Q</td>
<td>Quit</td>
</tr>
</tbody>
</table>

In prompt mode, you can issue the F command alone to inject a single fault in a random location in the device. In the following examples using the F command in prompt mode, three errors are injected.

F #3 0x12 0x34 0x56 0x78 * 0x9A 0xBC +
- Error 1 – Single bit error at frame 0x12, bit 0x34
- Error 2 – Uncorrectable error at frame 0x56, bit 0x78 (an * indicates a multi-bit error)
- Error 3 – Double-adjacent error at frame 0x9A, bit 0xBC (a + indicates a double bit error)

F 0x12 0x34 0x56 0x78 *
One (default) error is injected:
Error 1 – Single bit error at frame 0x12, bit 0x34. Locations after the first frame/bit location are ignored.

F #3 0x12 0x34 0x56 0x78 * 0x9A 0xBC 0xDE 0x00
Three errors are injected:
- Error 1 – Single bit error at frame 0x12, bit 0x34
- Error 2 – Uncorrectable error at frame 0x56, bit 0x78
- Error 3 – Double-adjacent error at frame 0x9A, bit 0xBC
- Locations after the first 3 frame/bit pairs are ignored

Determining CRAM Bit Locations

When the Fault Injection Debugger detects a CRAM EDCRC error, the Error Message Register (EMR) contains the syndrome, frame number, bit location, and error type (single, double, or multi-bit) of the detected CRAM error.

During system testing, save the EMR contents reported by the Fault Injection Debugger when you detect an EDCRC fault.

Note: With the recorded EMR contents, you can supply the frame and bit numbers to the Fault Injection Debugger to replay the errors noted during system testing, to further design, and characterize a system recovery response to that error.

Related Information

AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices

Advanced Command-Line Options: ASD Regions and Error Type Weighting

You can use the Fault Injection Debugger command-line interface to inject errors into ASD regions and weight the error types.
First, you specify the mix of error types (single bit, double adjacent, and multi-bit uncorrectable) using the `--weight <single errors>,<double adjacent errors>,<multi-bit errors>` option. For example, for a mix of 50% single errors, 30% double adjacent errors, and 20% multi-bit uncorrectable errors, use the option `--weight=50.30.20`. Then, to target an ASD region, use the `-smh` option to include the SMH file and indicate the ASD region to target. For example:

```
$ quartus_fid --cable=USB-BlasterII --index "@1=svgx.sof#pi" --weight=100.0.0 --smh="@1=svgx.smh#2" --number=30
```

This example command:
- Programs the device and injects faults (`pi` string)
- Injects 100% single-bit faults (100.0.0)
- Injects only into `ASD_REGION 2` (indicated by the #2)
- Injects 30 faults

### Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
</tbody>
</table>
| 2015.05.04    | 15.0.0  | • Provided more detail on how to use the Fault Injection Debugger throughout the document.  
            |         | • Added more command-line examples.                                      |
| 2014.06.30    | 14.0.0  | • Removed “Modifying the Quartus INI File” section.                     
            |         | • Added “Targeted Fault Injection Feature” section.                     
            |         | • Updated “Hardware and Software Requirements” section.                 |
| December 2012 | 2012.12.01 | Preliminary release.                                                  |
In-System Debugging Using External Logic Analyzers

About the Quartus Prime Logic Analyzer Interface

The Quartus Prime Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Altera-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Altera®-supported device.

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Quartus Prime LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your Altera-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the Quartus Prime LAI enables you to map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Quartus Prime LAI.

Note: The term “logic analyzer” when used in this document includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The LAI does not support Hard Processor System (HPS) I/Os.

Related Information
Device Support website

Choosing a Logic Analyzer

The Quartus Prime software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The SignalTap® II Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Altera-supported device by using the Quartus Prime LAI
## Table 11-1: Comparing the SignalTap II Logic Analyzer with the Logic Analyzer Interface

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Recommended Logic Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Depth</td>
<td>You have access to a wider sample depth with an external logic analyzer. In the SignalTap II Logic Analyzer, the maximum sample depth is set to 128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.</td>
<td>LAI</td>
</tr>
<tr>
<td>Debugging Timing Issues</td>
<td>Using an external logic analyzer provides you with access to a “timing” mode, which enables you to debug combined streams of data.</td>
<td>LAI</td>
</tr>
<tr>
<td>Performance</td>
<td>You frequently have limited routing resources available to place and route when you use the SignalTap II Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.</td>
<td>LAI</td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.</td>
<td>LAI or SignalTap II</td>
</tr>
<tr>
<td>Use of Output Pins</td>
<td>Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.</td>
<td>SignalTap II</td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>With the SignalTap II Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.</td>
<td>SignalTap II</td>
</tr>
</tbody>
</table>

### Related Information
- **System Debugging Tools Overview** on page 5-1
  Overview and comparison of all tools available in the Quartus Prime software on-chip debugging tool suite

### Required Components
You must have the following components to perform analysis using the LAI:
• The Quartus Prime software starting with version 5.1 and later
• The device under test
• An external logic analyzer
• An Altera communications cable
• A cable to connect the Altera-supported device to the external logic analyzer

Figure 11-1: LAI and Hardware Setup

Notes to figure:
1. Configuration and control of the LAI using a computer loaded with the Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.
Flow for Using the LAI

Figure 11-2: LAI Workflow

Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

Working with LAI Files

The .lai file stores the configuration of an LAI instance. The .lai file opens in the LAI editor. The editor allows you to group multiple internal signals to a set of external pins.

Configuring the File Core Parameters

After you create the .lai file, you must configure the .lai file core parameters by clicking on the Setup View list, and then selecting Core Parameters. The table below lists the .lai file core parameters.

Table 11-2: LAI File Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Count</td>
<td>The Pin Count parameter signifies the number of pins you want dedicated to your LAI. The pins must be connected to a debug header on your board. Within the Altera-supported device, each pin is mapped to a user-configurable number of internal signals. The Pin Count parameter can range from 1 to 255 pins.</td>
</tr>
</tbody>
</table>
**Parameter**  |  **Description**  
--- | ---  
**Bank Count**  |  The **Bank Count** parameter signifies the number of internal signals that you want to map to each pin. For example, a **Bank Count** of 8 implies that you will connect eight internal signals to each pin. The **Bank Count** parameter can range from 1 to 255 banks. 

**Output/Capture Mode**  |  The **Output/Capture Mode** parameter signifies the type of acquisition you perform. There are two options that you can select:  

- **Combinational/Timing**—This acquisition uses your external logic analyzer’s internal clock to determine when to sample data. Because **Combinational/Timing** acquisition samples data asynchronously to your Altera-supported device, you must determine the sample frequency you should use to debug and verify your system. This mode is effective if you want to measure timing information, such as channel-to-channel skew. For more information about the sampling frequency and the speeds at which it can run, refer to the data sheet for your external logic analyzer.  

- **Registered/State**—This acquisition uses a signal from your system under test to determine when to sample. Because **Registered/State** acquisition samples data synchronously with your Altera-supported device, it provides you with a functional view of your Altera-supported device while it is running. This mode is effective when you verify the functionality of your design. 

**Clock**  |  The **Clock** parameter is available only when **Output/Capture Mode** is set to **Registered State**. You must specify the sample clock in the **Core Parameters** view. The sample clock can be any signal in your design. However, for best results, Altera recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire. 

**Power-Up State**  |  The **Power-Up State** parameter specifies the power-up state of the pins you have designated for use with the LAI. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled. 

---

**Mapping the LAI File Pins to Available I/O Pins**

To configure the .lai file I/O pin parameters, select **Pins** in the **Setup View** list. To assign pin locations for the LAI, double-click the **Location** column next to the reserved pins in the **Name** column, and the Pin Planner opens.

**Related Information**

- Managing Device I/O Pins documentation  
- Information about how to use the Pin Planner

**Mapping Internal Signals to the LAI Banks**

After you have specified the number of banks to use in the **Core Parameters** settings page, you must assign internal signals for each bank in the LAI. Click the **Setup View** arrow and select **Bank n** or **All Banks**.

To view all of your bank connections, click **Setup View** and select **All Banks**.
Using the Node Finder

Before making bank assignments, on the View menu, point to Utility Windows and click Node Finder. Find the signals that you want to acquire, then drag and drop the signals from the Node Finder dialog box into the bank Setup View. When adding signals, use SignalTap II: pre-synthesis for non-incrementally routed instances and SignalTap II: post-fitting for incrementally routed instances.

As you continue to make assignments in the bank Setup View, the schematic of your LAI in the Logical View of your .lai file begins to reflect your assignments. Continue making assignments for each bank in the Setup View until you have added all of the internal signals for which you wish to acquire data.

Compiling Your Quartus Prime Project

When you save your .lai file, a dialog box prompts you to enable the LAI instance for the active project. Alternatively, you can specify the .lai file your project uses in the Global Project Settings dialog box.

After you specify the name of your .lai file, you must compile your project. To compile your project, on the Processing menu, click Start Compilation.

To ensure that the LAI is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to Utility Windows and click Project Navigator.) If the LAI is compiled with your design, the sld_hub and sld_multitap entities are shown in the Project Navigator.

Figure 11-3: Project Navigator

Programming Your Altera-Supported Device Using the LAI

After compilation completes, you must configure your Altera-supported device before using the LAI. You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Altera, JTAG-compliant devices. To use the LAI in more than one Altera-supported device, create an .lai file and configure an .lai file for each Altera-supported device that you want to analyze.

Controlling the Active Bank During Runtime

When you have programmed your Altera-supported device, you can control which bank you map to the reserved .lai file output pins. To control which bank you map, in the schematic in the Logical View, right-click the bank and click Connect Bank.
Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer. For more information about this process and for guidelines about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

Document Revision History

Table 11-3: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Dita conversion&lt;br&gt;• Added limitation about HPS I/O support</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.1.1</td>
<td>Changed to new document template</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Minor editorial updates&lt;br&gt;• Changed to new document template</td>
</tr>
<tr>
<td>August 2010</td>
<td>10.0.1</td>
<td>Corrected links</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Created links to the Quartus Prime Help&lt;br&gt;• Editorial updates&lt;br&gt;• Removed Referenced Documents section</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Removed references to APEX devices&lt;br&gt;• Editorial updates</td>
</tr>
</tbody>
</table>
### Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| March 2009    | 9.0.0   | • Minor editorial updates  
• Removed Figures 15-4, 15-5, and 15-11 from 8.1 version            |
| November 2008 | 8.1.0   | Changed to 8-1/2 x 11 page size. No change to content                   |
| May 2008      | 8.0.0   | • Updated device support list on page 15-3  
• Added links to referenced documents throughout the chapter  
• Added “Referenced Documents”  
• Added reference to Section V. In-System Debugging  
• Minor editorial updates |

**Related Information**

**Quartus Handbook Archive**
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
In-System Modification of Memory and Constants

About the In-System Memory Content Editor

The Quartus Prime In-System Memory Content Editor allows you to view and update memories and constants with the JTAG port connection.

The In-System Memory Content Editor allows access to dense and complex FPGA designs. When you program devices, you have read and write access to the memories and constants through the JTAG interface. You can then identify, test, and resolve issues with your design by testing changes to memory contents in the FPGA while your design is running.

When you use the In-System Memory Content Editor in conjunction with the SignalTap II Logic Analyzer, you can more easily view and debug your design in the hardware lab.

The ability to read data from memories and constants allows you to quickly identify the source of problems. The write capability allows you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Memory Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. You can also intentionally write incorrect parity bit values into your RAM to check the error handling functionality of your design.

Related Information

System Debugging Tools Overview on page 5-1
Overview and comparison of all tools available in the Quartus Prime software on-chip debugging tool suite

Design Debugging Using the SignalTap II Logic Analyzer documentation on page 9-1

Library of Parameterized Modules online help
List of the types of memories and constants currently supported by the Quartus Prime software

Design Flow Using the In-System Memory Content Editor

To use the In-System Memory Content Editor, perform the following steps:
1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.

Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the Quartus Prime software changes the default implementation. A single-port RAM is converted to a dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design.

If you instantiate a memory or constant IP core directly with ports and parameters in VHDL or Verilog HDL, add or modify the lpm_hint parameter as follows:

In VHDL code, add the following:

```vhdl
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
            INSTANCE_NAME = <instantiation name>";
```

In Verilog HDL code, add the following:

```verilog
defparam <megafunction instance name>.lpm_hint =
    "ENABLE_RUNTIME_MOD = YES,
     INSTANCE_NAME = <instantiation name>";
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor has three separate panes: the Instance Manager, the JTAG Chain Configuration, and the Hex Editor.

The Instance Manager pane displays all available run-time modifiable memories and constants in your FPGA device. The JTAG Chain Configuration pane allows you to program your FPGA and select the Altera® device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the specific device selected in the JTAG Chain Configuration pane.

If you have more than one device with in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus Prime software to access the memories and constants in each of the devices. Each In-System Memory Content Editor can access the in-system memories and constants in a single device.

Instance Manager

When you scan the JTAG chain to update the Instance Manager pane, you can view a list of all run-time modifiable memories and constants in the design. The Instance Manager pane displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.
You can read and write to in-system memory with the **Instance Manager** pane.

**Note:** In addition to the buttons available in the **Instance Manager** pane, you can read and write data by selecting commands from the Processing menu, or the right-click menu in the **Instance Manager** pane or **Hex Editor** pane.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running**, **Offloading data**, or **Updating data**. The health monitor provides information about the status of the editor.

The Quartus Prime software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the **In-System Memory Content Editor Settings** section of the Compilation Report to match an index number with the corresponding instance ID.

**Related Information**

*Instance Manager Pane online help*

---

**Editing Data Displayed in the Hex Editor Pane**

You can edit data read from your in-system memories and constants displayed in the **Hex Editor** pane by typing values directly into the editor or by importing memory files.

**Importing and Exporting Memory Files**

The In-System Memory Content Editor allows you to import and export data values for memories that have the In-System Updating feature enabled. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file enables you to save the contents of the memory for future use.

**Scripting Support**

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered at a command prompt. For detailed information about scripting command options, refer to the Quartus Prime command-line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```
The commonly used commands for the In-System Memory Content Editor are as follows:

- **Reading from memory:**
  ```
  read_content_from_memory
  [-content_in_hex]
  -instance_index <instance index>
  -start_address <starting address>
  -word_count <word count>
  ```

- **Writing to memory:**
  ```
  write_content_to_memory
  ```

- **Saving memory contents to a file:**
  ```
  save_content_from_memory_to_file
  ```

- **Updating memory contents from a file:**
  ```
  update_content_to_memory_from_file
  ```

**Related Information**

- Tcl Scripting documentation
- Command-Line Scripting documentation
- API Functions for Tcl online help
  Descriptions of the command options and scripting examples

**Programming the Device with the In-System Memory Content Editor**

If you make changes to your design, you can program the device from within the In-System Memory Content Editor.

**Example: Using the In-System Memory Content Editor with the SignalTap II Logic Analyzer**

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Logic Analyzer to efficiently debug your design. You can use the In-System Memory Content Editor and the SignalTap II Logic Analyzer simultaneously with the JTAG interface.

**Scenario:** After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the SignalTap II Logic Analyzer.
2. Using the SignalTap II Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cutoff frequency.
3. Using the In-System Memory Content Editor, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.
4. Because your coefficients are in-system modifiable, you update the coefficients with the correct data with the In-System Memory Content Editor.

In this scenario, you can quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II Logic Analyzer. You can also verify the functionality of your device by changing the coefficient values before modifying the design source files.
You can also modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter, for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function.

### Document Revision History

#### Table 12-1: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Dita conversion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed references to megafUNCTION and replaced with IP core.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.0.3</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.2</td>
<td>Changed to new document template. No change to content.</td>
</tr>
<tr>
<td>August 2010</td>
<td>10.0.1</td>
<td>Corrected links</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Inserted links to Quartus Prime Help</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Reference Documents section</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Delete references to APEX devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Style changes</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>No change to content</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Added reference to Section V. In-System Debugging in volume 3 of the Quartus Prime Handbook on page 16-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed references to the Mercury device, as it is now considered to be a “Mature” device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added links to referenced documents throughout document</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run time. The SignalTap II Logic Analyzer and SignalProbe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design.

You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

- Force the occurrence of trigger conditions set up in the SignalTap II Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the Quartus Prime software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the SignalTap II Logic Analyzer or SignalProbe, the In-System Sources and Probes Editor gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG IP core and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Quartus Prime software offers a variety of on-chip debugging tools.

The In-System Sources and Probes Editor consists of the ALTSOURCE_PROBE IP core and an interface to control the ALTSOURCE_PROBE IP core instances during run time. Each ALTSOURCE_PROBE IP core instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. When you compile your design, the ALTSOURCE_PROBE IP core sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE_PROBE IP core instances. The figure shows a block diagram of the components that make up the In-System Sources and Probes Editor.
The ALTSOURCE_PROBE IP core hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. If used in conjunction with the SignalTap II Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly

The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE_PROBE IP core instances to increase the level of automation.
Related Information

System Debugging Tools
For an overview and comparison of all the tools available in the Quartus Prime software on-chip debugging tool suite

Hardware and Software Requirements
The following components are required to use the In-System Sources and Probes Editor:

- Quartus Prime software
- Quartus Prime Lite Edition (with the TalkBack feature turned on)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® development kit or user design board with a JTAG connection to device under test

The In-System Sources and Probes Editor supports the following device families:

- Arria® series
- Stratix® series
- Cyclone® series
- MAX® series

Design Flow Using the In-System Sources and Probes Editor
The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the In-System Sources and Probes IP core.

After you compile the design, you can control each instance via the In-System Sources and Probes Editor pane or via a Tcl interface.
Instantiating the In-System Sources and Probes IP Core

You must instantiate the In-System Sources and Probes IP core before you can use the In-System Sources and Probes editor. Use the IP Catalog and parameter editor to instantiate a custom variation of the In-System Sources and Probes IP core.
To configure the In-System Sources and Probes IP core, perform the following steps:

1. On the Tools menu, click **Tools > IP Catalog**.
2. Locate and double-click the In-System Sources and Probes IP core. The parameter editor appears.
3. Specify a name for your custom IP variation.
4. Specify the desired parameters for your custom IP variation. You can specify up to up to 512 bits for each source. Your design may include up to 128 instances of this IP core.
5. Click **Generate** or **Finish** to generate IP core synthesis and simulation files matching your specifications. The parameter editor generates the necessary variation files and the instantiation template based on your specification. Use the generated template to instantiate the In-System Sources and Probes IP core in your design.

**Note:** The In-System Sources and Probes Editor does not support simulation. You must remove the In-System Sources and Probes IP core before you create a simulation netlist.

### In-System Sources and Probes IP Core Parameters

Use the template to instantiate the variation file in your design.

#### Table 13-1: In-System Sources and Probes IP Port Information

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required?</th>
<th>Direction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe[]</td>
<td>No</td>
<td>Input</td>
<td>The outputs from your design.</td>
</tr>
<tr>
<td>source_clk</td>
<td>No</td>
<td>Input</td>
<td>Source Data is written synchronously to this clock. This input is required if you turn on <strong>Source Clock</strong> in the <strong>Advanced Options</strong> box in the parameter editor.</td>
</tr>
<tr>
<td>source_ena</td>
<td>No</td>
<td>Input</td>
<td>Clock enable signal for source_clk. This input is required if specified in the <strong>Advanced Options</strong> box in the parameter editor.</td>
</tr>
<tr>
<td>source[]</td>
<td>No</td>
<td>Output</td>
<td>Used to drive inputs to user design.</td>
</tr>
</tbody>
</table>

You can include up to 128 instances of the in-system sources and probes IP core in your design, if your device has available resources. Each instance of the IP core uses a pair of registers per signal for the width of the widest port in the IP core. Additionally, there is some fixed overhead logic to accommodate communication between the IP core instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

When you compile your design that includes the In-System Sources and Probes IP core, the In-System Sources and Probes and SLD Hub Controller IP core are added to your compilation hierarchy automatically. These IP cores provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the In-System Sources and Probes IP core. To open the design instance you want to modify in the parameter editor, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

### Compiling the Design

When you compile your design that includes the In-System Sources and Probes IP core, the In-System Sources and Probes and SLD Hub Controller IP core are added to your compilation hierarchy...
automatically. These IP cores provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the In-System Sources and Probes IP core. To open the design instance you want to modify in the parameter editor, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

**Running the In-System Sources and Probes Editor**

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE IP core instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE IP core in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor:

- On the **Tools** menu, click **In-System Sources and Probes Editor**.

**In-System Sources and Probes Editor GUI**

The In-System Sources and Probes Editor contains three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Quartus Prime software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE_PROBE IP core by scanning the JTAG chain and sending a query to the device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane can access the ALTSOURCE_PROBE IP core instances in a single device. If you have more than one device containing IP core instances in a JTAG chain, you can launch multiple **In-System Sources and Probes Editor** panes to access the IP core instances in each device.

**Programming Your Device With JTAG Chain Configuration**

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor.
To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the JTAG Chain Configuration pane, point to Hardware, and then select the hardware communications device. You may be prompted to configure your hardware; in this case, click Setup.
3. From the Device list, select the FPGA device to which you want to download the design (the device may be automatically detected). You may need to click Scan Chain to detect your target device.
4. In the JTAG Chain Configuration pane, click to browse for the SRAM Object File (.sof) that includes the In-System Sources and Probes instance or instances. (The .sof may be automatically detected).
5. Click Program Device to program the target device.

### Instance Manager

The Instance Manager pane provides a list of all ALTSOURCE_PROBE instances in the design and allows you to configure how data is acquired from or written to those instances.

The following buttons and sub-panes are provided in the Instance Manager pane:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the In-System Sources and Probes Editor pane.
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the In-System Sources and Probes Editor pane; you can modify the sample rate via the Probe read interval setting.
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.
- **Write Source Data**—Writes data to all source nodes of the selected instance.
- **Probe Read Interval**—Displays the sample interval of all the In-System Sources and Probe instances in your design; you can modify the sample interval by clicking Manual.
- **Event Log**—Controls the event log in the In-System Sources and Probes Editor pane.
- **Write Source Data**—Allows you to manually or continuously write data to the system.

The status of each instance is also displayed beside each entry in the Instance Manager pane. The status indicates if the instance is Not running Offloading data, Updating data, or if an Unexpected JTAG communication error occurs. This status indicator provides information about the sources and probes instances in your design.

### In-System Sources and Probes Editor Pane

The In-System Sources and Probes Editor pane allows you to view data from all sources and probes in your design.

The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

### Reading Probe Data

You can read data by selecting the ALTSOURCE_PROBE instance in the Instance Manager pane and clicking Read Probe Data.
This action produces a single sample of the probe data and updates the data column of the selected index in the In-System Sources and Probes Editor pane. You can save the data to an event log by turning on the Save data to event log option in the Instance Manager pane.

If you want to sample data from your probe instance continuously, in the Instance Manager pane, click the instance you want to read, and then click Continuously read probe data. While reading, the status of the active instance shows Unloading. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the Instance Manager pane.

To adjust the probe read interval, in the Instance Manager pane, turn on the Manual option in the Probe read interval sub-pane, and specify the sample rate in the text field next to the Manual option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the Current interval box. You can adjust the event log window buffer size in the Maximum Size box.

### Writing Data

To modify the source data you want to write into the ALTSOURCE_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive out to the ALTSOURCE_PROBE instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer.

Modified values that are not written out to the ALTSOURCE_PROBE instances appear in red. To update the ALTSOURCE_PROBE instance, highlight the instance in the Instance Manager pane and click Write source data. The Write source data function is also available via the shortcut menus in the Instance Manager pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE_PROBE instances. To continuously update the ALTSOURCE_PROBE instances, change the Write source data field from Manually to Continuously.

### Organizing Data

The In-System Sources and Probes Editor pane allows you to group signals into buses, and also allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select Group. You can modify the display format in the Bus Display Format and the Bus Bit order shortcut menus.

The In-System Sources and Probes Editor pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (.spf). To save changes, on the File menu, click Save. The file contains all the modifications you made to the signal groups, as well as the current data event log.
Tcl interface for the In-System Sources and Probes Editor

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run `quartus_stp`.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a custom command set.

Table 13-2: In-System Sources and Probes Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start_insystem_source_probe</code></td>
<td><code>-device_name &lt;device name&gt;</code></td>
<td>Opens a handle to a device with the specified hardware.</td>
</tr>
<tr>
<td></td>
<td><code>-hardware_name &lt;hardware name&gt;</code></td>
<td>Call this command before starting any transactions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>get_insystem_source_probe_instance_info</code></td>
<td><code>-device_name &lt;device name&gt;</code></td>
<td>Returns a list of all <code>ALTSOURCE_PROBE</code> instances in your design. Each record returned is in the following format:</td>
</tr>
<tr>
<td></td>
<td><code>-hardware_name &lt;hardware name&gt;</code></td>
<td><code>{&lt;instance Index&gt;, &lt;source width&gt;, &lt;probe width&gt;, &lt;instance name&gt;}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>read_probe_data</code></td>
<td><code>-instance_index &lt;instance_index&gt;</code></td>
<td>Retrieves the current value of the probe.</td>
</tr>
<tr>
<td></td>
<td><code>-value_in_hex (optional)</code></td>
<td>A string is returned that specifies the status of each probe, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>read_source_data</code></td>
<td><code>-instance_index &lt;instance_index&gt;</code></td>
<td>Retrieves the current value of the sources.</td>
</tr>
<tr>
<td></td>
<td><code>-value_in_hex (optional)</code></td>
<td>A string is returned that specifies the status of each source, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>write_source_data</code></td>
<td><code>-instance_index &lt;instance_index&gt;</code></td>
<td>Sets the value of the sources.</td>
</tr>
<tr>
<td></td>
<td><code>-value &lt;value&gt;</code></td>
<td>A binary string is sent to the source ports, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td><code>-value_in_hex (optional)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>end_interactive_probe</code></td>
<td>None</td>
<td>Releases the JTAG chain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Issue this command when all transactions are finished.</td>
</tr>
</tbody>
</table>
The example shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in the figure below. The example design contains a DCFIFO with ALTSOURCE_PROBE instances to read from and write to the DCFIFO. A set of control muxes are added to the design to control the flow of data to the DCFIFO between the input pins and the ALTSOURCE_PROBE instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The ALTSOURCE_PROBE instances, when used with the script in the example below, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

Use the Tcl script in debugging situations to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the SignalTap II Logic Analyzer.

Figure 13-3: DCFIFO Example Design Controlled by Tcl Script

```tcl
## Setup USB hardware - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain
set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure: argument value is integer
proc write {value} {
    global device_name usb
    variable full
    start_insystem_source_probe -device_name $device_name -hardware_name $usb 
    #read full flag
    start_insystem_source_probe -device_name $device_name -hardware_name $usb 
    #read full flag
```
Related Information

- Tcl Scripting
- Quartus Prime Settings File Manual
- Command Line Scripting

Design Example: Dynamic PLL Reconfiguration

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.

Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG IP core provides an easy interface to access the register chain counters. The ALTPLL_RECONFIG IP core provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG IP core drives the PLL register chain to update the PLL with the updated parameters. The figure shows a Stratix-enhanced PLL with reconfigurable coefficients.
The following design example uses an ALTSOURCE_PROBE instance to update the PLL parameters in the ALTPLL_RECONFIG IP core cache. The ALTPLL_RECONFIG IP core connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new m and n values for the enhanced PLL. The Tcl script extracts the m and n values from the GUI, shifts the values out to the ALTSOURCE_PROBE instances to update the values in the ALTPLL_RECONFIG IP core cache, and asserts the reconfiguration signal on the ALTPLL_RECONFIG IP core. The reconfiguration signal on the ALTPLL_RECONFIG IP core starts the register chain transaction to update all PLL reconfigurable coefficients.
This design example was created using a Nios® II Development Kit, Stratix Edition. The file `sourceprobe_DE_dynamic_pll.zip` contains all the necessary files for running this design example, including the following:

- **Readme.txt**—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in the figure below.
- **Interactive_Reconfig.qar**—The archived Quartus Prime project for this design example.

**Related Information**

- [On-chip Debugging Design Examples](#) to download the In-System Sources and Probes Example

---

**Document Revision History**

**Table 13-3: Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>Updated formatting.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2011</td>
<td>10.1.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Minor corrections. Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Minor corrections.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Removed references to obsolete devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Style changes.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Documented that this feature does not support simulation on page 17–5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 17–8 for Interactive PLL reconfiguration manager</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Prime Handbook Archive**
For previous versions of the Quartus Prime Handbook
The Quartus Prime Programmer allows you to program and configure Altera® CPLD, FPGA, and configuration devices. After compiling your design, use the Quartus Prime Programmer to program or configure your device, to test the functionality of the design on a circuit board.

Related Information
- Programming Devices

Programming Flow

Figure 14-1: Programming Flow
The following steps describe the programming flow:

1. Compile your design, such that the Quartus Prime Assembler generates the programming or configuration file.
2. Convert the programming or configuration file to target your configuration device and, optionally, create secondary programming files.

### Table 14-1: Programming and Configuration File Format

<table>
<thead>
<tr>
<th>File Format</th>
<th>FPGA</th>
<th>CPLD</th>
<th>Configuration Device</th>
<th>Serial Configuration Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM Object File (.sof)</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Programmer Object File (.pof)</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JEDEC JESD71 STAPL Format File (.jam)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>Jam Byte Code File (.jbc)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
</tbody>
</table>

3. Program and configure the FPGA, CPLD, or configuration device using the programming or configuration file with the Quartus Prime Programmer.

### Figure 14-2: Programming File Generation Flow

**Stand-Alone Quartus Prime Programmer**

Altera offers the free stand-alone Programmer, which has the same full functionality as the Quartus Prime Programmer in the Quartus Prime software. The stand-alone Programmer is useful when programming your devices with another workstation, so you do not need two full licenses. You can download the stand-alone Programmer from the Download Center on the Altera website.
Stand-Alone Programmer Memory Limitations

The stand-alone Programmer may use significant memory during the following operations:

- During auto-detect operations
- When the programming file is added to the flash
- During manual attachment of the flash into the Programmer window

The 32-bit stand-alone Programmer can only use a limited amount of memory when launched in 32-bit Windows. Note the following specific limitations of 32-bit stand-alone Programmer:

Table 14-2: Stand-Alone Programmer Memory Limitations

<table>
<thead>
<tr>
<th>Application</th>
<th>Maximum Flash Device Size</th>
<th>Flash Device Operation Using PFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Stand-Alone Programmer</td>
<td>Up to 512 Mb</td>
<td>Single Flash Device</td>
</tr>
<tr>
<td>64-bit Stand-Alone Programmer</td>
<td>Up to 2 Gb</td>
<td>Multiple Flash Device</td>
</tr>
</tbody>
</table>

The stand-alone Programmer supports combination and/or conversion of Quartus Prime programming files using the Convert Programming Files dialog box. You can convert programming files, such as Mask Settings File (.msf), Partial-Mask SRAM Object File (.pmsf), SRAM Object Files (.sof), or Programmer Object Files (.pof) into other file formats that support device configuration schemes for Altera devices.

Related Information
- [Download Center](#)
  You can download the stand-alone Quartus Prime Programmer from this page.

Optional Programming or Configuration Files

The Quartus Prime software can generate optional programming or configuration files in various formats that you can use with programming tools other than the Quartus Prime Programmer. When you compile a design in the Quartus Prime software, the Assembler automatically generates either a .sof or .pof. The Assembler also allows you to convert FPGA configuration files to programming files for configuration devices.

Related Information
- [AN 425: Using Command-Line Jam STAPL Solution for Device Programming](#)
  Describes how to use the .jam and .jbc programming files with the Jam STAPL Player, Jam STAPL Byte-Code Player, and the quartus_jli command-line executable.

Secondary Programming Files

The Quartus Prime software generates programming files in various formats for use with different programming tools.

Table 14-3: File Types Generated by the Quartus Prime Software and Supported by the Quartus Prime Programmer

<table>
<thead>
<tr>
<th>File Type</th>
<th>Generated by the Quartus Prime Software</th>
<th>Supported by the Quartus Prime Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>.sof</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Quartus Prime Programmer GUI

The Quartus Prime Programmer GUI is a window that allows you to perform the following tasks:

- Adding your programming and configuration files.
- Specifying programming options and hardware.
- Starting the programming or configuration of the device.

To open the Programmer window, on the Tools menu, click Programmer. As you proceed through the programming flow, the Quartus Prime Message window reports the status of each operation.

#### Related Information

- [Programmer Page (Options Dialog Box)](#)
  Describes the options in the Tools menu.

#### Editing the Device Details of an Unknown Device

If the Quartus Prime Programmer automatically detects devices with shared JTAG IDs, the Programmer prompts you to specify the correct device in the JTAG chain.

If the Programmer does not prompt you to specify the correct device in the JTAG chain, then you must add a user defined device in the Quartus Prime software for each unknown device in the JTAG chain and specify the instruction register length for each device.

---

(8) Raw Binary File (.rbf) is supported by the Quartus Prime Programmer in Passive Serial (PS) configuration mode.
To edit the device details of an unknown device, follow these steps:

1. Double-click on the unknown device listed under the device column.
2. Click **Edit**.
3. Change the device **Name**.
4. Enter the **Instruction register Length**.
5. Click **OK**.
6. Save the `.cdf`.

**Setting Up Your Hardware**

The Quartus Prime Programmer provides the flexibility to choose a download cable or programming hardware. Before you can program or configure your device, you must have the correct hardware setup.

**Related Information**

- **Setting up Programming Hardware in Quartus Prime Software**
  Describes the programming hardware driver installation.

**Setting the JTAG Hardware**

The JTAG server allows the Quartus Prime Programmer to access the JTAG hardware. You can also access the JTAG download cable or programming hardware connected to a remote computer through the JTAG server of that computer. With the JTAG server, you can control the programming or configuration of devices from a single computer through other computers at remote locations. The JTAG server uses the TCP/IP communications protocol.

**Running JTAG Daemon with Linux**

The JTAGD daemon is the Linux version of a JTAG server. The JTAGD daemon allows a board which is connected to a Linux host to be programmed or debugged over the network from a remote machine. The JTAGD daemon also allows multiple programs to use JTAG resources at the same time.

Run the JTAGD daemon to avoid:

- the JTAGD server from exiting after two minutes of idleness.
- the JTAGD server from not accepting connections from remote machines, which might lead to an intermittent failure.

To run JTAGD as a daemon, follow these steps:

1. Create an `/etc/jtagd` directory.
2. Set the permissions of this directory and the files in the directory to allow you to have the read/write access.
3. Run `jtagd` (with no arguments) from your `quartus/bin` directory.

The JTAGD daemon is now running and does not terminate when you log off.

**Using the JTAG Chain Debugger Tool**

The JTAG Chain Debugger tool allows you to test the JTAG chain integrity and detect intermittent failures of the JTAG chain. In addition, the tool allows you to shift in JTAG instructions and data through
the JTAG interface and step through the test access port (TAP) controller state machine for debugging purposes. You access the tool from the Tools menu on the main menu of the Quartus Prime software.

Programming and Configuration Modes

The following table lists the programming and configuration modes supported by Altera devices.

Table 14-4: Programming and Configuration Modes

<table>
<thead>
<tr>
<th>Configuration Mode Supported by the Quartus Prime Programmer</th>
<th>FPGA</th>
<th>CPLD</th>
<th>Configuration Device</th>
<th>Serial Configuration Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>Passive Serial (PS)</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Active Serial (AS) Programming</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>Configuration via Protocol (CvP)</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>In-Socket Programming</td>
<td>—</td>
<td>Yes (except for MAX II CPLDs)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Related Information

- **Configuration via Protocol (CvP) Implementation in Altera FPGAs User Guide**
  Describes the CvP configuration mode.
- **Programming Adapters**
  Contains a list of programming adapters available for Altera devices.

Design Security Keys

The Quartus Prime Programmer supports the generation of encryption key programming files and encrypted configuration files for Altera FPGAs that support the design security feature. You can also use the Quartus Prime Programmer to program the encryption key into the FPGA.

Related Information

- **AN 556: Using the Design Security Features in Altera FPGAs**

Convert Programming Files Dialog Box

The Convert Programming Files dialog box in the Programmer allows you to convert programming files from one file format to another. For example, to store the FPGA data in configuration devices, you can convert the .sof data to another format, such as .pof, .hexout, .rbf, .rdp, or .jic, and then program the configuration device.

You can also configure multiple devices with an external host, such as a microprocessor or CPLD. For example, you can combine multiple .sof files into one .pof. To save time in subsequent conversions, you can click **Save Conversion Setup** to save your conversion specifications in a Conversion Setup File (.cof).
Click **Open Conversion Setup Data** to load your .cof setup in the **Convert Programming Files** dialog box.

To access the **Convert Programming Files** dialog box, on the main menu of the Quartus Prime software, click **File > Convert Programming Files**.

**Example 14-1: Conversion Setup File Contents**

```xml
<?xml version="1.0" encoding="US-ASCII" standalone="yes"?>
<cof>
  <output_filename>output_file.pof</output_filename>
  <n_pages>1</n_pages>
  <width>1</width>
  <mode>14</mode>
  <sof_data>
    <user_name>Page_0</user_name>
    <page_flags>1</page_flags>
    <bit>
      <sof_filename>/users/jbrossar/template/output_files/template_test.sof</sof_filename>
    </bit>
  </sof_data>
  <version>7</version>
  <create_cvp_file>0</create_cvp_file>
  <create_hps_iocsr>0</create_hps_iocsr>
  <auto_create_rpd>0</auto_create_rpd>
  <options>
    <map_file>1</map_file>
  </options>
  <MAX10_device_options>
    <por>0</por>
    <io_pullup>1</io_pullup>
    <auto_reconfigure>1</auto_reconfigure>
    <isp_source>0</isp_source>
    <verify_protect>0</verify_protect>
    <epof>0</epof>
    <ufm_source>0</ufm_source>
  </MAX10_device_options>
  <advanced_options>
    <ignore_epcs_id_check>0</ignore_epcs_id_check>
    <ignore_condone_check>2</ignore_condone_check>
    <plc_adjustment>0</plc_adjustment>
    <post_chain_bitstream_pad_bytes>-1</post_chain_bitstream_pad_bytes>
    <post_device_bitstream_pad_bytes>-1</post_device_bitstream_pad_bytes>
  </advanced_options>
</cof>
```

**Related Information**

- **Convert Programming Files Dialog Box**

**Debugging Your Configuration**

Use the **Advanced** option in the **Convert Programming Files** dialog box to debug your configuration. You must choose the advanced settings that apply to your Altera device. You can direct the Quartus Prime software to enable or disable an advanced option by turning the option on or off in the **Advanced Options** dialog box.
When you change settings in the Advanced Options dialog box, the change affects .pof, .jic, .rdp, and .rbf files.

The following table lists the Advanced Options settings in more detail.

### Table 14-5: Advanced Options Settings

<table>
<thead>
<tr>
<th>Option Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable EPCS ID check</td>
<td>FPGA skips the EPCS silicon ID verification. Default setting is unavailable (EPCS ID check is enabled). Applies to the single- and multi-device AS configuration modes on all FPGA devices.</td>
</tr>
<tr>
<td>Disable AS mode CONF_DONE error check</td>
<td>FPGA skips the CONF_DONE error check. Default setting is unavailable (AS mode CONF_DONE error check is enabled). Applies to single- and multi-device (AS) configuration modes on all FPGA devices. The CONF_DONE error check is disabled by default for Stratix V, Arria V, and Cyclone V devices for AS-PS multi device configuration mode.</td>
</tr>
<tr>
<td>Program Length Count adjustment</td>
<td>Specifies the offset you can apply to the computed PLC of the entire bitstream. Default setting is 0. The value must be an integer. Applies to single- and multi-device (AS) configuration modes on all FPGA devices.</td>
</tr>
<tr>
<td>Post-chain bitstream pad bytes</td>
<td>Specifies the number of pad bytes appended to the end of an entire bitstream. Default value is set to 0 if the bitstream of the last device is uncompressed. Set to 2 if the bitstream of the last device is compressed.</td>
</tr>
<tr>
<td>Post-device bitstream pad bytes</td>
<td>Specifies the number of pad bytes appended to the end of the bitstream of a device. Default value is 0. No negative integer. Applies to all single-device configuration modes on all FPGA devices.</td>
</tr>
<tr>
<td>Option Setting</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bitslice padding value</td>
<td>Specifies the padding value used to prepare bitslice configuration bitstreams, such that all bitslice configuration chains simultaneously receive their final configuration data bit. Default value is 1. Valid setting is 0 or 1. Use only in 2, 4, and 8-bit PS configuration mode, when you use an EPC device with the decompression feature enabled. Applies to all FPGA devices that support enhanced configuration devices.</td>
</tr>
</tbody>
</table>

The following table lists the symptoms you may encounter if a configuration fails, and describes the advanced options you must use to debug your configuration.

<table>
<thead>
<tr>
<th>Failure Symptoms</th>
<th>Disable EPCS ID Check</th>
<th>Disable AS Mode CONF_DONE Error Check</th>
<th>PLC Settings</th>
<th>Post-Chain Bitstream Pad Bytes</th>
<th>Post-Device Bitstream Pad Bytes</th>
<th>Bitslice Padding Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration failure occurs after a configuration cycle.</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (9)</td>
<td>Yes (10)</td>
<td>—</td>
</tr>
<tr>
<td>Decompression feature is enabled.</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (9)</td>
<td>Yes (10)</td>
<td>—</td>
</tr>
<tr>
<td>Encryption feature is enabled.</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (9)</td>
<td>Yes (10)</td>
<td>—</td>
</tr>
<tr>
<td>CONF_DONE stays low after a configuration cycle.</td>
<td>—</td>
<td>Yes</td>
<td>Yes (11)</td>
<td>Yes (9)</td>
<td>Yes (10)</td>
<td>—</td>
</tr>
</tbody>
</table>

(9) Use only for multi-device chain
(10) Use only for single-device chain
(11) Start with positive offset to the PLC settings
### Failure Symptoms

<table>
<thead>
<tr>
<th>Failure Symptoms</th>
<th>Disable EPCS ID Check</th>
<th>Disable AS Mode CONF_DONE Error Check</th>
<th>PLC Settings</th>
<th>Post-Chain Bitstream Pad Bytes</th>
<th>Post-Device Bitstream Pad Bytes</th>
<th>Bitslice Padding Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONF_DONE goes high momentarily after a configuration cycle.</td>
<td>—</td>
<td>Yes</td>
<td>Yes&lt;sup&gt;(12)&lt;/sup&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FPGA does not enter user mode even though CONF_DONE goes high.</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes&lt;sup&gt;(9)&lt;/sup&gt;</td>
<td>Yes&lt;sup&gt;(10)&lt;/sup&gt;</td>
<td>—</td>
</tr>
<tr>
<td>Configuration failure occurs at the beginning of a configuration cycle.</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Newly introduced EPCS, such as EPCS128.</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Failure in .pof generation for EPC device using Quartus Prime Convert Programming File Utility when the decompression feature is enabled.</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<sup>(12)</sup> Start with negative offset to the PLC settings
Flash Loaders

Parallel and serial configuration devices do not support the JTAG interface. However, you can use a flash loader to program configuration devices in-system via the JTAG interface. You can use an FPGA as a bridge between the JTAG interface and the configuration device. The Quartus Prime software supports parallel and serial flash loaders.

Scripting Support

In addition to the Quartus Prime Programmer GUI, you can use the Quartus Prime command-line executable `quartus_pgm.exe` to access programmer functionality from the command line and from scripts. The programmer accepts `.pof`, `.sof`, and `.jic` programming or configuration files and `.cdf`.

The following example shows a command that programs a device:

```
quartus_pgm -c byteblasterII -m jtag -o bpv;design.pof
```

Where:

- `-c byteblasterII` specifies the ByteBlaster II download cable
- `-m jtag` specifies the JTAG programming mode
- `-o bpv` represents the blank-check, program, and verify operations
- `design.pof` represents the `.pof` used for the programming

The Programmer automatically executes the erase operation before programming the device.

**Note:** For linux terminal, use the following command:

```
quartus_pgm -c byteblasterII -m jtag -o bpv\;design.pof
```

Related Information

- About Quartus Prime Scripting

The jtagconfig Debugging Tool

You can use the `jtagconfig` command-line utility (which is similar to the auto detect operation in the Quartus Prime Programmer) to check the devices in a JTAG chain and the user-defined devices.

For more information about the `jtagconfig` utility, type one of the following commands at the command prompt:

```
jtagconfig -h
```

```
jtagconfig --help
```

**Note:** The help switch does not reference the `-n` switch. The `jtagconfig -n` command shows each node for each JTAG device.

Related Information

Command-Line Scripting
Generating .pmsf using a .msf and a .sof

You can generate a .pmsf with the quartus_cpf command by typing the following command:

```
quartus_cpf -p <pr_revision.msf> <pr_revision.sof> <new_filename.pmsf>
```

### Document Revision History

#### Table 14-6: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>Added Conversion Setup File (.cof) description and example.</td>
</tr>
<tr>
<td>December 2014</td>
<td>14.1.0</td>
<td>Updated the Scripting Support section to include a Linux command to program a device.</td>
</tr>
<tr>
<td>June 2014</td>
<td>14.0.0</td>
<td>• Added Running JTAG Daemon.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed Cyclone III and Stratix III devices references.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed MegaWizard Plug-In Manager references.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Secondary Programming Files section to add notes about the Quartus Prime Programmer support for .rbf files.</td>
</tr>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>• Converted to DITA format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added JTAG Debug Mode for Partial Reconfiguration and Configuring Partial Reconfiguration Bitstream in JTAG Debug Mode sections.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Updated Table 18–3 on page 18–6, and Table 18–4 on page 18–8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added “Converting Programming Files for Partial Reconfiguration” on page 18–10, “Generating .pmsf using a .msf and a .sof” on page 18–10, “Generating .rbf for Partial Reconfiguration Using a .pmsf” on page 18–12, “Enable Decompression during Partial Reconfiguration Option” on page 18–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Scripting Support” on page 18–15.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Updated Table 18–5 on page 18–8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Quartus Prime Programmer GUI” on page 18–3.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.1.0</td>
<td>• Updated “Configuration Modes” on page 18–5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added “Optional Programming or Configuration Files” on page 18–6.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Table 18–2 on page 18–5.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>• Added links to Quartus Prime Help.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “Hardware Setup” on page 21–4 and “JTAG Chain Debugger Tool” on page 21–4.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated “JTAG Chain Debugger Example” on page 20–4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added links to Quartus Prime Help.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reorganized chapter.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Added links to Quartus Prime Help.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deleted screen shots.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Added a row to Table 21–4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed references from “JTAG Chain Debug” to “JTAG Chain Debugger”.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated figures.</td>
</tr>
</tbody>
</table>

**Related Information**

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
You can integrate your supported EDA simulator into the Quartus Prime design flow. This chapter provides specific guidelines for simulation of Quartus Prime designs with the Aldec Active-HDL or Riviera-PRO software.

Quick Start Example (Active-HDL VHDL)

You can adapt the following RTL simulation example to get started quickly with Active-HDL:

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:
   
   ```
   set_user_option -name EDA_TOOL_PATH_ACTIVEHDL <Active HDL executable path>
   set_global_assignment -name EDA_SIMULATION_TOOL "Active-HDL (VHDL)"
   ```

2. Compile simulation model libraries using one of the following methods:
   
   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
   - Compile Altera simulation models manually:
     ```
     vlib <library1> <altera_library1>
     vcom -strict93 -dbg -work <library1> <lib1_component/pack.vhd> <lib1.vhd>
     ```

3. Create and open the workspace:
   ```
   createdesign <workspace name> <workspace path>
   opendesign -a <workspace name>.adf
   ```

4. Create the work library and compile the netlist and testbench files:
   ```
   vlib work
   vcom -strict93 -dbg -work work <output netlist> <testbench file>
   ```

5. Load the design:
   ```
   vsim +access+r -t 1ps +transport_int_delays +transport_path_delays \ 
   -L work -L <lib1> -L <lib2> work.<testbench module name>
   ```

6. Run the simulation in the Active-HDL simulator.
Aldec Active-HDL and Riviera-PRO Guidelines

The following guidelines apply to simulating Altera designs in the Active-HDL or Riviera-PRO software.

Compiling SystemVerilog Files

If your design includes multiple SystemVerilog files, you must compile the System Verilog files together with a single alog command. If you have Verilog files and SystemVerilog files in your design, you must first compile the Verilog files, and then compile only the SystemVerilog files in the single alog command.

Simulating Transport Delays

By default, the Active-HDL or Riviera-PRO software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the in the Active-HDL or Riviera-PRO software prevents the simulator from filtering out these pulses.

Table 15-1: Transport Delay Simulation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+transport_path_delays</td>
<td>Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the +pulse_e/number and +pulse_r/number options.</td>
</tr>
<tr>
<td>+transport_int_delays</td>
<td>Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the +pulse_int_e/number and +pulse_int_r/number options.</td>
</tr>
</tbody>
</table>

Note: The +transport_path_delays and +transport_path_delays options apply automatically during NativeLink gate-level timing simulation.

To perform a gate-level timing simulation with the device family library, type the Active-HDL command:

vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo \ work.filtref_vhd_vec_tst +transport_int_delays +transport_path_delays

Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing.

Before you begin

By default, the x_on_violation_option logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the x_on_violation_option logic option for the specific register, as shown in the following example from the Quartus Prime Settings File (.qsf).

set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \<register_name>
Using Simulation Setup Scripts

The Quartus Prime software can generate a rivierapro_setup.tcl simulation setup script for IP cores in your design. The use and content of the script file is similar to the msim_setup.tcl file used by the ModelSim simulator.

Document Revision History

Table 15-2: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of Quartus II to Quartus Prime.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>• Replaced MegaWizard Plug-In Manager information with IP Catalog.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Relocated general simulation information to Simulating Altera Designs.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>• Changed to new document template.</td>
</tr>
</tbody>
</table>

Related Information

Quartus Handbook Archive
For previous versions of the Quartus Prime Handbook, refer to the Quartus Handbook Archive.
You can integrate your supported EDA simulator into the Quartus Prime design flow. This document provides guidelines for simulation of Quartus Prime designs with the Synopsys VCS or VCS MX software.

Quick Start Example (VCS with Verilog)

You can adapt the following RTL simulation example to get started quickly with VCS:

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:
   
   ```
   set_user_option -name EDA_TOOL_PATH_VCS <VCS executable path>
   
   set_global_assignment -name EDA_SIMULATION_TOOL "VCS"
   ```

2. Compile simulation model libraries using one of the following methods:
   
   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.

3. Modify the `simlib_comp.vcs` file to specify your design and testbench files.

4. Type the following to run the VCS simulator:
   
   ```
   vcs -R -file simlib_comp.vcs
   ```

VCS and QuestaSim Guidelines

The following guidelines apply to simulation of Altera designs in the VCS or VCS MX software:

- Do not specify the `-v` option for `altera_lnsim.sv` because it defines a systemverilog package.
- Add `+verilog and +verilog2001ext+.v` options to make sure all `.v` files are compiled as verilog 2001 files, and all other files are compiled as systemverilog files.
- Add the `-lca` option for Stratix V and later families because they include IEEE-encrypted simulation files for VCS and VCS MX.
- Add `-timescale=1ps/1ps` to ensure picosecond resolution.
Simulating Transport Delays

By default, the VCS and VCS MX software filter out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the VCS and VCS MX software prevents the simulator from filtering out these pulses.

### Table 16-1: Transport Delay Simulation Options (VCS and VCS MX)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+transport_path_delays</td>
<td>Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the +pulse_e/number and +pulse_r/number options.</td>
</tr>
<tr>
<td>+transport_int_delays</td>
<td>Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the +pulse_int_e/number and +pulse_int_r/number options.</td>
</tr>
</tbody>
</table>

**Note:** The +transport_path_delays and +transport_path_delays options apply automatically during NativeLink gate-level timing simulation.

The following VCS and VCS MX software command runs a post-synthesis simulation:

```bash
vcs -R <testbench>.v <gate-level netlist>.v -v <Altera device family library>.v +transport_int_delays +pulse_int_e/0 +pulse_int_r/0 +transport_path_delays +pulse_e/0 +pulse_r/0
```

Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing.

**Before you begin**

By default, the `x_on_violation_option` logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the `x_on_violation_option` logic option for the specific register, as shown in the following example from the Quartus Prime Settings File (.qsf).

```bash
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \ <register_name>
```

Generating Power Analysis Files

You can generate a Verilog Value Change Dump File (.vcd) for power analysis in the Quartus Prime software, and then run the `.vcd` from the VCS software. Use this `.vcd` for power analysis in the Quartus Prime PowerPlay power analyzer.
Before you begin

To generate and use a .vcd for power analysis, follow these steps:

1. In the Quartus Prime software, click Assignments > Settings.
2. Under EDA Tool Settings, click Simulation.
3. Turn on Generate Value Change Dump file script, specify the type of output signals to include, and specify the top-level design instance name in your testbench.
4. Click Processing > Start Compilation.
5. Use the following command to include the script in your testbench where the design under test (DUT) is instantiated:
   `include <revision_name>_dump_all_vcd_nodes.v`

   **Note:** Include the script within the testbench module block. If you include the script outside of the testbench module block, syntax errors occur during compilation.
6. Run the simulation with the VCS command. Exit the VCS software when the simulation is finished and the `<revision_name>.vcd` file is generated in the simulation directory.

**VCS Simulation Setup Script Example**

The Quartus Prime software can generate a simulation setup script for IP cores in your design. The scripts contain shell commands that compile the required simulation models in the correct order, elaborate the top-level design, and run the simulation for 100 time units by default. You can run these scripts from a Linux command shell.

The scripts for VCS and VCS MX are `vcs_setup.sh` (for Verilog HDL or SystemVerilog) and `vcsmx_setup.sh` (combined Verilog HDL and SystemVerilog with VHDL). Read the generated `.sh` script to see the variables that are available for override when sourcing the script or redefining directly if you edit the script. To set up the simulation for a design, use the command-line to pass variable values to the shell script.

**Example 16-1: Using Command-line to Pass Simulation Variables**

```bash
sh vcsmx_setup.sh
USER_DEFINED_ELAB_OPTIONS=+rad
USER_DEFINED_SIM_OPTIONS=+vcs+lic+wait
```

**Example 16-2: Example Top-Level Simulation Shell Script for VCS-MX**

```bash
# Run generated script to compile libraries and IP simulation files
# Skip elaboration and simulation of the IP variation
sh ./ip_top_sim/synopsys/vcsmx/vcsmx_setup.sh SKIP_ELAB=1 SKIP_SIM=1 QSYS_SIMDIR="./ip_top_sim"

# Compile top-level testbench that instantiates IP
vlogan -sverilog ./top_testbench.sv

# Elaborate and simulate the top-level design
vcs -lca -t ps <elaboration control options> top_testbench
simv <simulation control options>
```
Example 16-3: Example Top-Level Simulation Shell Script for VCS

```bash
# Run script to compile libraries and IP simulation files
sh ./ip_top_sim/synopsys/vcs/vcs_setup.sh TOP_LEVEL_NAME="top_testbench"
# Pass VCS elaboration options to compile files and elaborate top-level
# passed to the script as the TOP_LEVEL_NAME
USER_DEFINED_ELAB_OPTIONS="top_testbench.sv"
# Pass in simulation options and run the simulation for specified amount of
time.
USER_DEFINED_SIM_OPTIONS="<simulation control options>
```

Document Revision History

Table 16-2: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>• Replaced MegaWizard Plug-In Manager information with IP Catalog.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Relocated general simulation information to Simulating Altera Designs.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>• Changed to new document template.</td>
</tr>
</tbody>
</table>

Related Information

*Quartus Handbook Archive*

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
You can integrate a supported EDA simulator into the Quartus Prime design flow. This document provides guidelines for simulation of designs with Mentor Graphics® ModelSim-Altera®, ModelSim, or QuestaSim software. Altera provides the entry-level ModelSim-Altera software, along with precompiled Altera simulation libraries, to simplify simulation of Altera designs.

**Note:** The latest version of the ModelSim-Altera software supports native, mixed-language (VHDL/Verilog HDL/SystemVerilog) co-simulation of plain text HDL. If you have a VHDL-only simulator, you can use the ModelSim-Altera software to simulate Verilog HDL modules and IP cores. Alternatively, you can purchase separate co-simulation software.

**Related Information**
- Simulating Altera Designs on page 1-1
- Managing Quartus Prime Projects

### Quick Start Example (ModelSim with Verilog)

You can adapt the following RTL simulation example to get started quickly with ModelSim:

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:

   ```
   set_user_option -name EDA_TOOL_PATH_MODELSIM <modelsim executable path>
   set_global_assignment -name EDA_SIMULATION_TOOL "MODELSIM (verilog)"
   ```

2. Compile simulation model libraries using one of the following methods:
• Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
• Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
• Type the following commands to create and map Altera simulation libraries manually, and then compile the models manually:

```
vlib <lib1>_ver
vmap <lib1>_ver <lib1>_ver
vlog -work <lib1> <lib1>
```

3. Compile your design and testbench files:

```
vlog -work work <design or testbench name>.v
```

4. Load the design:

```
sim -L work -L <lib1>_ver -L <lib2>_ver work.<testbench name>
```

**ModelSim, ModelSim-Altera, and QuestaSim Guidelines**

The following guidelines apply to simulation of Altera designs in the ModelSim, ModelSim-Altera, or QuestaSim software.

**Using ModelSim-Altera Precompiled Libraries**

Precompiled libraries for both functional and gate-level simulations are provided for the ModelSim-Altera software. You should not compile these library files before running a simulation. No precompiled libraries are provided for ModelSim or QuestaSim. You must compile the necessary libraries to perform functional or gate-level simulation with these tools.

The precompiled libraries provided in `<ModelSim-Altera path>/altera/` must be compatible with the version of the Quartus Prime software that creates the simulation netlist. To verify compatibility of precompiled libraries with your version of the Quartus Prime software, refer to the `<ModelSim-Altera path>/altera/version.txt` file. This file indicates the Quartus Prime software version and build of the precompiled libraries.

**Note:** Encrypted Altera simulation model files shipped with the Quartus Prime software version 10.1 and later can only be read by ModelSim-Altera Edition Software version 6.6c and later. These encrypted simulation model files are located at the `<Quartus Prime System directory>/quartus/eda/sim_lib/mentor` directory.

**Disabling Timing Violation on Registers**

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing.

**Before you begin**

By default, the `x_on_violation_option` logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register,
disable the \texttt{x\_on\_violation\_option} logic option for the specific register, as shown in the following example from the Quartus Prime Settings File (.qsf).

\begin{verbatim}
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \ <register_name>
\end{verbatim}

**Passing Parameter Information from Verilog HDL to VHDL**

You must use in-line parameters to pass values from Verilog HDL to VHDL.

**Before you begin**

By default, the \texttt{x\_on\_violation\_option} logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the \texttt{x\_on\_violation\_option} logic option for the specific register, as shown in the following example from the Quartus Prime Settings File (.qsf).

\begin{verbatim}
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \ <register_name>
\end{verbatim}

**Example 17-1: In-line Parameter Passing Example**

\begin{verbatim}
lpm_add_sub#(.lpm_width(12), .lpm_direction("Add"),
    .lpm_type("LPM_ADD_SUB"),
    .lpm_hint("ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO") )

lpm_add_sub_component (
    .dataa (dataa),
    .datab (datab),
    .result (sub_wire0)
);
\end{verbatim}

\textbf{Note:} The sequence of the parameters depends on the sequence of the GENERIC in the VHDL component declaration.

**Increasing Simulation Speed**

By default, the ModelSim and QuestaSim software runs in a debug-optimized mode.

**Before you begin**

To run the ModelSim and QuestaSim software in speed-optimized mode, add the following two vlog command-line switches. In this mode, module boundaries are flattened and loops are optimized, which eliminates levels of debugging hierarchy and may result in faster simulation. This switch is not supported in the ModelSim-Altera simulator.

\begin{verbatim}
vlog -fast -05
\end{verbatim}

**Simulating Transport Delays**

By default, the ModelSim and QuestaSim software filter out all pulses that are shorter than the propagation delay between primitives.

Turning on the \texttt{transport delay} options in the ModelSim and QuestaSim software prevents the simulator from filtering out these pulses.
### Table 17-1: Transport Delay Simulation Options (ModelSim and QuestaSim)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+transport_path_delays</td>
<td>Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the +pulse_e/number and +pulse_r/number options.</td>
</tr>
<tr>
<td>+transport_int_delays</td>
<td>Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the +pulse_int_e/number and +pulse_int_r/number options.</td>
</tr>
</tbody>
</table>

**Note:** The +transport_path_delays and +transport_path_delays options apply automatically during NativeLink gate-level timing simulation. For more information about either of these options, refer to the ModelSim-Altera Command Reference installed with the ModelSim and QuestaSim software.

The following ModelSim and QuestaSim software command shows the command line syntax to perform a gate-level timing simulation with the device family library:

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo work.filtref_vhd_vec_tst \
+transport_int_delays +transport_path_delays
```

### Viewing Error Messages

ModelSim and QuestaSim error and warning messages are tagged with a vsim or vcom code. To determine the cause and resolution for a vsim or vcom error or warning, use the verror command.

For example, ModelSim may return the following error:

```
# ** Error: C:/altera_trn/DUALPORT_TRY/simulation/modelsim/DUALPORT_TRY.vho(31):
    (vcom-1136) Unknown identifier "stratixiv"
```

In this case, type the following command:

```
verror 1136
```

The following description appears:

```
# vcom Message # 1136:
# The specified name was referenced but was not found. This indicates
# that either the name specified does not exist or is not visible at
# this point in the code.
```

### Generating Power Analysis Files

To generate a timing Value Change Dump File (.vcd) for power analysis, you must first generate a `<filename>_dump_all_vcd_nodes.tcl` script file in the Quartus Prime software. You can then run the script from the ModelSim, QuestaSim, or ModelSim-Altera software to generate a timing `<filename>.vcd` for use in the Quartus Prime PowerPlay power analyzer.
Before you begin

To generate and use a .vcd for power analysis, follow these steps:

1. In the Quartus Prime software, click Assignments > Settings.
2. Under EDA Tool Settings, click Simulation.
3. Turn on Generate Value Change Dump file script, specify the type of output signals to include, and specify the top-level design instance name in your testbench.
4. Click Processing > Start Compilation.
5. Click Tools > Run EDA Simulation > EDA Gate Level Simulation. The Compiler creates the <filename>_dump_all_vcd_nodes.tcl file, the ModelSim simulation <filename>_run_msim_gate_vhdl/verilog.do file (including the .vcd and .tcl execution lines), and all other files for simulation. ModelSim then automatically runs the generated .do to start the simulation.
6. Stop the simulation if your testbench does not have a break point. ModelSim generates the .vcd only after simulation ends with the End Simulation function.

Viewing Simulation Waveforms

ModelSim-Altera, ModelSim, and QuestaSim automatically generate a Wave Log Format File (.wlf) following simulation. You can use the .wlf to generate a waveform view.

Before you begin

To view a waveform from a .wlf through ModelSim-Altera, ModelSim, or QuestaSim, perform the following steps:

1. Type vsim at the command line. The ModelSim/QuestaSim or ModelSim-Altera dialog box appears.
2. Click File > Datasets. The Datasets Browser dialog box appears.
3. Click Open and select your .wlf.
4. Click Done.
5. In the Object browser, select the signals that you want to observe.
6. Click Add > Wave, and then click Selected Signals.
   You must first convert the .vcd to a .wlf before you can view a waveform in ModelSim-Altera, ModelSim, or QuestaSim.
7. To convert the the .vcd to a .wlf, type the following at the command-line:

   vcd2wlf <example>.vcd <example>.wlf

8. After conversion, view the .wlf waveform in ModelSim or QuestaSim.
   You can convert your .wlf to a .vcd by using the wlf2vcd command

Simulating with ModelSim-Altera Waveform Editor

You can use the ModelSim-Altera Waveform Editor as a simple method to create stimulus vectors for simulation. You can create this design stimulus via interactive manipulation of waveforms from the wave window in ModelSim-Altera. With the ModelSim-Altera waveform editor, you can create and edit waveforms, drive simulation directly from created waveforms, and save created waveforms into a stimulus file.

Related Information

ModelSim Web Page
ModelSim Simulation Setup Script Example

The Quartus Prime software can generate a `msim_setup.tcl` simulation setup script for IP cores in your design. The script compiles the required device library models, compiles the design files, and elaborates the design with or without simulator optimization. To run the script, type source `msim_setup.tcl` in the simulator Transcript window.

Alternatively, if you are using the simulator at the command line, you can type the following command:

```
vsim -c -do msim_setup.tcl
```

In this example the `top-level-simulate.do` custom top-level simulation script sets the hierarchy variable `TOP_LEVEL_NAME` to `top_testbench` for the design, and sets the variable `QSYS_SIMDIR` to the location of the generated simulation files.

```
# Set hierarchy variables used in the IP-generated files
set TOP_LEVEL_NAME "top_testbench"
set QSYS_SIMDIR "./ip_top_sim"
# Source generated simulation script which defines aliases used below
source $QSYS_SIMDIR/mentor/msim_setup.tcl
# dev_com alias compiles simulation libraries for device library files
dev_com
# com alias compiles IP simulation or Qsys model files and/or Qsys model files in the correct order
com
# Compile top level testbench that instantiates your IP
vlog -sv ./top_testbench.sv
# elab alias elaborates the top-level design and testbench
elab
# Run the full simulation
run - all
```

In this example, the top-level simulation files are stored in the same directory as the original IP core, so this variable is set to the IP-generated directory structure. The `QSYS_SIMDIR` variable provides the relative hierarchy path for the generated IP simulation files. The script calls the generated `msim_setup.tcl` script and uses the alias commands from the script to compile and elaborate the IP files required for simulation along with the top-level simulation testbench. You can specify additional simulator elaboration command options when you run the `elab` command, for example, `elab +nowarnTFMPC`. The last command run in the example starts the simulation.

Unsupported Features

The Quartus Prime software does not support the following ModelSim simulation features:

- Altera does not support companion licensing for ModelSim AE.
- The USB software guard is not supported by versions earlier than Mentor Graphics ModelSim software version 5.8d.
- For ModelSim-Altera software versions prior to 5.5b, use the `PCLS` utility included with the software to set up the license.
- Some versions of ModelSim and QuestaSim support SystemVerilog, PSL assertions, SystemC, and more. For more information about specific feature support, refer to Mentor Graphics literature.

Related Information

- ModelSim-Altera Software Web Page
## Document Revision History

### Table 17-2: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2015.05.04</td>
<td>15.0.0</td>
<td>• Added mixed language simulation support in the ModelSim-Altera software.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>• Replaced MegaWizard Plug-In Manager information with IP Catalog.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Relocated general simulation information to Simulating Altera Designs.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>• Changed to new document template.</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.
You can integrate your supported EDA simulator into the Quartus Prime design flow. This chapter provides specific guidelines for simulation of Quartus Prime designs with the Cadence Incisive Enterprise (IES) software.

Quick Start Example (NC-Verilog)

You can adapt the following RTL simulation example to get started quickly with IES:

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:
   
   ```
   set_user_option -name EDA_TOOL_PATH_NCSIM <ncsim executable path>
   
   set_global_assignment -name EDA_SIMULATION_TOOL "NC-Verilog (Verilog)"
   ```

2. Compile simulation model libraries using one of the following methods:
   
   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
   - Map Altera simulation libraries by adding the following commands to a cds.lib file:
     
     ```
     include ${CDS_INST_DIR}/tools/inca/files/101/files/cds.lib
     DEFINE <lib1> <lib1_ver>
     ```

   Then, compile Altera simulation models manually:

   ```
   vlog -work <lib1_ver>
   ```

3. Elaborate your design and testbench with IES:

   ```
   ncelab <work library>.<top-level entity name>
   ```

4. Run the simulation:

   ```
   ncsim <work library>.<top-level entity name>
   ```
Cadence Incisive Enterprise (IES) Guidelines

The following guidelines apply to simulation of Altera designs in the IES software:

- Do not specify the -v option for `altera_insim.sv` because it defines a systemverilog package.
- Add `-verilog` and `+verilog2001ext+.v` options to make sure all `.v` files are compiled as verilog 2001 files, and all other files are compiled as systemverilog files.
- Add the `-lca` option for Stratix V and later families because they include IEEE-encrypted simulation files for IES.
- Add `-timescale=1ps/1ps` to ensure picosecond resolution.

Using GUI or Command-Line Interfaces

Altera supports both the IES GUI and command-line simulator interfaces.

To start the IES GUI, type `nclaunch` at a command prompt.

Table 18-1: Simulation Executables

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncvlog</td>
<td><code>ncvlog</code> compiles your Verilog HDL code and performs syntax and static semantics checks.</td>
</tr>
<tr>
<td>ncvhdl</td>
<td><code>ncvhdle</code> compiles your VHDL code and performs syntax and static semantics checks.</td>
</tr>
<tr>
<td>ncelab</td>
<td>Elaborates the design hierarchy and determines signal connectivity.</td>
</tr>
<tr>
<td>ncsdfc</td>
<td>Performs back-annotation for simulation with VHDL simulators.</td>
</tr>
<tr>
<td>ncsim</td>
<td>Runs mixed-language simulation. This program is the simulation kernel that performs event scheduling and executes the simulation code.</td>
</tr>
</tbody>
</table>

Elaborating Your Design

The simulator automatically reads the `.sdo` file during elaboration of the Quartus Prime-generated Verilog HDL or SystemVerilog HDL netlist file. The `ncelab` command recognizes the embedded system task `$sdf_annotate` and automatically compiles and annotates the `.sdo` file by running `ncsdfc` automatically.

VHDL netlist files do not contain system task calls to locate your `.sdf` file; therefore, you must compile the standard `.sdo` file manually. Locate the `.sdo` file in the same directory where you run elaboration or simulation. Otherwise, the `$sdf_annotate` task cannot reference the `.sdo` file correctly. If you are starting an elaboration or simulation from a different directory, you can either comment out the `$sdf_annotate` and annotate the `.sdo` file with the GUI, or add the full path of the `.sdo` file.

Note: If you use NC-Sim for post-fit VHDL functional simulation of a Stratix V design that includes RAM, an elaboration error might occur if the component declaration parameters are not in the same order as the architecture parameters. Use the `-namemap_mixgen` option with the `ncelab` command to match the component declaration parameter and architecture parameter names.
Back-Annotating Simulation Timing Data (VHDL Only)

You can back annotate timing information in a Standard Delay Output File (.sdo) for VHDL simulators. To back annotate the .sdo timing data at the command line, follow these steps:

1. To compile the .sdo with the ncsdfc program, type the following command at the command prompt. The ncsdfc program generates an <output name>.sdf.X compiled .sdo file

   ncsdfc <project name>_vhd.sdo -output <output name>

   **Note:** If you do not specify an output name, ncsdfc uses <project name>.sdo.X

2. Specify the compiled .sdf file for the project by adding the following command to an ASCII SDF command file for the project:

   COMPILED_SDF_FILE = "<project name>.sdf.X" SCOPE = <instance path>

3. After compiling the .sdf file, type the following command to elaborate the design:

   ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File>

Example 18-1: Example SDF Command File

```plaintext
// SDF command file sdf_file
COMPILED_SDF_FILE = "lpm_ram_dp_test_vhd.sdo.X",
SCOPE = :tb,
MTM_CONTROL = "TYPICAL",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing.

**Before you begin**

By default, the x_on_violation_option logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the x_on_violation_option logic option for the specific register, as shown in the following example from the Quartus Prime Settings File (.qsf).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \<register_name>
```

Simulating Pulse Reject Delays

By default, the IES software filters out all pulses that are shorter than the propagation delay between primitives.

Setting the pulse reject delays options in the IES software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.
Table 18-2: Pulse Reject Delay Options

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-PULSE_R</td>
<td>Use when simulation pulses are shorter than the delay in a gate-level primitive. The argument is the percentage of delay for pulse reject limit for the path</td>
</tr>
<tr>
<td>-PULSE_INT_R</td>
<td>Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. The argument is the percentage of delay for pulse reject limit for the path</td>
</tr>
</tbody>
</table>

Viewing Simulation Waveforms

IES generates a .trn file automatically following simulation. You can use the .trn for generating the SimVision waveform view.

Before you begin

To view a waveform from a .trn file through SimVision, follow these steps:

1. Type simvision at the command line. The Design Browser dialog box appears.
2. Click File > Open Database and click the .trn file.
3. In the Design Browser dialog box, select the signals that you want to observe from the Hierarchy.
4. Right-click the selected signals and click Send to Waveform Window.

You cannot view a waveform from a .vcd file in SimVision, and the .vcd file cannot be converted to a .trn file.

IES Simulation Setup Script Example

The Quartus Prime software can generate a ncsim_setup.sh simulation setup script for IP cores in your design. The script contains shell commands that compile the required device libraries, IP, or Qsys simulation models in the correct order. The script then elaborates the top-level design and runs the simulation for 100 time units by default. You can run these scripts from a Linux command shell. To set up the simulation script for a design, you can use the command-line to pass variable values to the shell script.

Read the generated .sh script to see the variables that are available for you to override when you source the script or that you can redefine directly in the generated .sh script. For example, you can specify additional elaboration and simulation options with the variables USER_DEFINED_ELAB_OPTIONS and USER_DEFINED_SIM_OPTIONS.

Example 18-2: Example Top-Level Simulation Shell Script for Incisive (NCSIM)

```bash
# Run script to compile libraries and IP simulation files
# Skip elaboration and simulation of the IP variation
sh ./ip_top_sim/cadence/ncsim_setup.sh SKIP_ELAB=1 SKIP_SIM=1 QSYS_SIMDIR="./ip_top_sim"

# Compile the top-level testbench that instantiates your IP
ncvlog -sv ./top_testbench.sv
# Elaborate and simulate the top-level design
```
## Document Revision History

### Table 18-3: Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015.11.02</td>
<td>15.1.0</td>
<td>Changed instances of <em>Quartus II</em> to <em>Quartus Prime</em>.</td>
</tr>
<tr>
<td>2014.08.18</td>
<td>14.0.a10.0</td>
<td>• Corrected incorrect references to VCS and VCS MX.</td>
</tr>
<tr>
<td>2014.06.30</td>
<td>14.0.0</td>
<td>• Replaced MegaWizard Plug-In Manager information with IP Catalog.</td>
</tr>
<tr>
<td>November 2012</td>
<td>12.1.0</td>
<td>• Relocated general simulation information to Simulating Altera Designs.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>• Changed to new document template.</td>
</tr>
</tbody>
</table>

### Related Information

**Quartus Handbook Archive**

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