Intel® Quartus® Prime Standard Edition User Guide

Platform Designer

Updated for Intel® Quartus® Prime Design Suite: 18.1
1. Creating a System with Platform Designer

1.1. Platform Designer Interface Support
1.2. Platform Designer System Design Flow
1.3. Starting or Opening a Project in Platform Designer
1.4. Viewing a Platform Designer System
   1.4.1. Viewing the System Hierarchy
   1.4.2. Filtering the System Contents
   1.4.3. Viewing Clock and Reset Domains
   1.4.4. Viewing Avalon Memory-Mapped Domains in a System
   1.4.5. Viewing the System Schematic
   1.4.6. Viewing System Assignments and Connections
   1.4.7. Customizing the Platform Designer Layout
1.5. Adding IP Components to a System
   1.5.1. Modifying IP Parameters
   1.5.2. Applying Preset Parameters for Specific Applications
   1.5.3. Adding Third-Party IP Components
   1.5.4. Creating or Opening an IP Core Variant
1.6. Connecting System Components
   1.6.1. Platform Designer 64-Bit Addressing Support
   1.6.2. Connecting Masters and Slaves
   1.6.3. Changing a Conduit to a Reset
   1.6.4. Previewing the System Interconnect
1.7. Specifying Interconnect Requirements
   1.7.1. Interconnect Requirements
1.8. Defining Instance Parameters
   1.8.1. Creating an Instance Parameter Script in Platform Designer
   1.8.2. Platform Designer Instance Parameter Script Tcl Commands
1.9. Implementing Performance Monitoring
1.10. Configuring Platform Designer System Security
   1.10.1. System Security Options
   1.10.2. Specifying a Default Slave
   1.10.3. Accessing Undefined Memory Regions
1.11. Upgrading Outdated IP Components
   1.11.1. Troubleshooting IP or Platform Designer System Upgrade
1.12. Synchronizing System Component Information
1.13. Generating a Platform Designer System
   1.13.1. Generation Dialog Box Options
   1.13.2. Specifying the Generation ID
   1.13.3. Files Generated for IP Cores and Platform Designer Systems
   1.13.4. Generating System Testbench Files
   1.13.5. Generating Example Designs for IP Components
   1.13.6. Generating the HPS IP Component System View Description File
   1.13.7. Generating Header Files for Master Components
1.14. Simulating a Platform Designer System
   1.14.1. Adding Assertion Monitors for Simulation
   1.14.2. Simulating Software Running on a Nios II Processor
1.15. Integrating a Platform Designer System with the Intel Quartus Prime Software
1.15.1. Integrate a Platform Designer System and the Intel Quartus Prime Software With the .qsys File .......................................................... 64
1.15.2. Integrate a Platform Designer System and the Intel Quartus Prime Software With the .qip File .................................................. 64
1.16. Managing Hierarchical Platform Designer Systems ................................................. 65
  1.16.1. Adding a Subsystem to a Platform Designer System ................................ 65
  1.16.2. Viewing and Traversing Subsystem Contents ............................................. 66
  1.16.3. Editing a Subsystem ............................................................................... 67
  1.16.4. Changing a Component's Hierarchy Level .............................................. 67
  1.16.5. Saving a Subsystem ............................................................................... 68
  1.16.6. Exporting a System as an IP Component ................................................. 68
  1.16.7. Hierarchical System Using Instance Parameters Example ....................... 69
1.17. Creating a System with Platform Designer Revision History ......................... 74

2. Optimizing Platform Designer System Performance ................................................. 76
  2.1. Designing with Avalon and AXI Interfaces ................................................... 76
    2.1.1. Designing Streaming Components ......................................................... 77
    2.1.2. Designing Memory-Mapped Components .............................................. 77
  2.2. Using Hierarchy in Systems ........................................................................... 78
  2.3. Using Concurrency in Memory-Mapped Systems .......................................... 81
    2.3.1. Implementing Concurrency With Multiple Masters ................................. 82
    2.3.2. Implementing Concurrency With Multiple Slaves ..................................... 83
    2.3.3. Implementing Concurrency with DMA Engines ....................................... 85
  2.4. Inserting Pipeline Stages to Increase System Frequency ............................... 86
  2.5. Using Bridges .............................................................................................. 87
    2.5.1. Using Bridges to Increase System Frequency ........................................... 88
    2.5.2. Using Bridges to Minimize Design Logic ............................................... 90
    2.5.3. Using Bridges to Minimize Adapter Logic .............................................. 92
    2.5.4. Considering the Effects of Using Bridges ............................................... 93
  2.6. Increasing Transfer Throughput ..................................................................... 99
    2.6.1. Using Pipelined Transfers ...................................................................... 100
    2.6.2. Arbitration Shares and Bursts ................................................................ 101
  2.7. Reducing Logic Utilization .......................................................................... 105
    2.7.1. Minimizing Interconnect Logic to Reduce Logic Utilization .................... 105
    2.7.2. Minimizing Arbitration Logic by Consolidating Multiple Interfaces ........ 106
    2.7.3. Reducing Logic Utilization With Multiple Clock Domains ...................... 108
    2.7.4. Duration of Transfers Crossing Clock Domains ....................................... 110
  2.8. Reducing Power Consumption ....................................................................... 111
    2.8.1. Reducing Power Consumption With Multiple Clock Domains ................. 111
    2.8.2. Reducing Power Consumption by Minimizing Toggle Rates .................... 114
    2.8.3. Reducing Power Consumption by Disabling Logic .................................. 115
  2.9. Reset Polarity and Synchronization in Platform Designer ............................... 116
  2.10. Optimizing Platform Designer System Performance Design Examples ........ 119
    2.10.1. Avalon Pipelined Read Master Example .............................................. 119
    2.10.2. Multiplexer Examples ......................................................................... 121

3. Platform Designer Interconnect .......................................................................... 124
  3.1. Memory-Mapped Interfaces ....................................................................... 124
    3.1.1. Platform Designer Packet Format ......................................................... 126
    3.1.2. Interconnect Domains ...................................................................... 129
    3.1.3. Master Network Interfaces ............................................................ 131
3.1.4. Slave Network Interfaces................................................................. 134
3.1.5. Arbitration.................................................................................... 136
3.1.6. Memory-Mapped Arbiter............................................................... 140
3.1.7. Datapath Multiplexing Logic.......................................................... 142
3.1.8. Width Adaptation.......................................................................... 142
3.1.9. Burst Adapter................................................................................ 144
3.1.10. Waitrequest Allowance Adapter.................................................. 146
3.1.11. Read and Write Responses......................................................... 147
3.1.12. Platform Designer Address Decoding........................................... 148
3.2. Avalon Streaming Interfaces............................................................... 148
3.2.1. Avalon-ST Adapters...................................................................... 150
3.3. Interrupt Interfaces........................................................................... 158
3.3.1. Individual Requests IRQ Scheme.................................................. 158
3.3.2. AssigningIRQs in Platform Designer.............................................. 159
3.4. Clock Interfaces................................................................................ 161
3.4.1. (High Speed Serial Interface) HSSI Clock Interfaces...................... 162
3.5. Reset Interfaces................................................................................ 167
3.5.1. Single Global Reset Signal Implemented by Platform Designer.......... 168
3.5.2. Reset Controller........................................................................... 168
3.5.3. Reset Bridge................................................................................ 168
3.5.4. Reset Sequencer........................................................................... 169
3.6. Conduits............................................................................................ 180
3.7. Interconnect Pipelining...................................................................... 180
3.7.1. Manually Control Pipelining in the Platform Designer Interconnect.... 183
3.8. Error Correction Coding (ECC) in Platform Designer Interconnect...... 184
3.9. AMBA 3 AXI Protocol Specification Support (version 1.0).................. 184
3.9.1. Channels...................................................................................... 184
3.9.2. Cache Support............................................................................. 185
3.9.3. Security Support.......................................................................... 186
3.9.4. Atomic Accesses......................................................................... 186
3.9.5. Response Signaling...................................................................... 186
3.9.6. Ordering Model........................................................................... 186
3.9.7. Data Buses.................................................................................. 187
3.9.8. Unaligned Address Commands.................................................... 187
3.9.9. Avalon and AXI Transaction Support............................................. 187
3.9.9.1. Avalon and AXI Transaction Support............................................. 187
3.10. AMBA 3 APB Protocol Specification Support (version 1.0)............... 188
3.10.1. Bridges...................................................................................... 188
3.10.2. Burst Adaptation........................................................................ 188
3.10.3. Width Adaptation....................................................................... 188
3.10.4. Error Response.......................................................................... 189
3.11. AMBA 4 AXI Memory-Mapped Interface Support (version 2.0)....... 189
3.11.1. Burst Support............................................................................ 189
3.11.2. QoS......................................................................................... 189
3.11.3. Regions.................................................................................... 189
3.11.4. Write Response Dependency...................................................... 189
3.11.5. AWACACHE and ARCACHE...................................................... 190
3.11.6. Width Adaptation and Data Packing in Platform Designer............ 190
3.11.7. Ordering Model........................................................................ 190
3.11.8. Read and Write Allocate............................................................ 190
3.11.9. Locked Transactions................................................................... 190
3.11.10. Memory Types........................................................................ 191
### Contents

3.11.11. Mismatched Attributes ................................................................. 191
3.11.12. Signals .......................................................................................... 191
3.12. AMBA 4 AXI Streaming Interface Support (version 1.0) ..................... 191
   3.12.1. Connection Points ........................................................................ 191
   3.12.2. Adaptation .................................................................................... 192
3.13. AMBA 4 AXI-Lite Protocol Specification Support (version 2.0) ........... 192
   3.13.1. AMBA 4 AXI-Lite Signals .............................................................. 192
   3.13.2. AMBA 4 AXI-Lite Bus Width ........................................................ 193
   3.13.3. AMBA 4 AXI-Lite Outstanding Transactions ................................. 193
   3.13.4. AMBA 4 AXI-Lite IDs .................................................................. 193
   3.13.5. Connections Between AMBA 3 AXI, AMBA 4 AXI and AMBA 4 AXI-Lite .................................................. 193
   3.13.6. AMBA 4 AXI-Lite Response Merging ........................................... 194
3.14. Port Roles (Interface Signal Types) ...................................................... 194
   3.14.1. AXI Master Interface Signal Types ................................................ 194
   3.14.2. AXI Slave Interface Signal Types .................................................. 195
   3.14.3. AMBA 4 AXI Master Interface Signal Types .................................. 196
   3.14.4. AMBA 4 AXI Slave Interface Signal Types .................................... 198
   3.14.5. AMBA 4 AXI-Stream Master and Slave Interface Signal Types .... 199
   3.14.6. ACE-Lite Interface Signal Roles ................................................... 200
   3.14.7. APB Interface Signal Types ......................................................... 200
   3.14.9. Avalon Streaming Interface Signal Roles ...................................... 204
   3.14.10. Avalon Clock Source Signal Roles ............................................... 205
   3.14.11. Avalon Clock Sink Signal Roles .................................................. 205
   3.14.15. Avalon Interrupt Sender Signal Roles ......................................... 208
3.15. Platform Designer Interconnect Revision History ................................. 208

4. Platform Designer System Design Components ......................................... 210
   4.1. Bridges ............................................................................................. 210
      4.1.1. Clock Bridge ............................................................................... 211
      4.1.2. Avalon-MM Clock Crossing Bridge ............................................. 212
      4.1.3. Avalon-MM Pipeline Bridge ......................................................... 214
      4.1.4. Avalon-MM Unaligned Burst Expansion Bridge ......................... 215
      4.1.5. Bridges Between Avalon and AXI Interfaces .............................. 218
      4.1.6. AXI Bridge ................................................................................. 219
      4.1.7. AXI Timeout Bridge .................................................................... 224
      4.1.8. Address Span Extender ............................................................... 228
   4.2. Error Response Slave ........................................................................ 233
      4.2.1. Error Response Slave Parameters ............................................. 234
      4.2.2. Error Response Slave CSR Registers ....................................... 235
      4.2.3. Designating a Default Slave ....................................................... 238
   4.3. Tri-State Components ....................................................................... 239
      4.3.1. Generic Tri-State Controller ...................................................... 241
      4.3.2. Tri-State Conduit Pin Sharer ...................................................... 241
      4.3.3. Tri-State Conduit Bridge ............................................................. 242
   4.4. Test Pattern Generator and Checker Cores ......................................... 242
      4.4.1. Test Pattern Generator ............................................................... 243
5.7.1. Specify HDL Files for Synthesis in the Platform Designer Component Editor........294
5.7.2. Analyze Synthesis Files in the Platform Designer Component Editor..............295
5.7.3. Name HDL Signals for Automatic Interface and Type Recognition in the
Platform Designer Component Editor...............................................................296
5.7.4. Specify Files for Simulation in the Component Editor........................................297
5.7.5. Include an Internal Register Map Description in the .svd for Slave
Interfaces Connected to an HPS Component..................................................298
5.8. Add Signals and Interfaces in the Platform Designer Component Editor......................299
5.9. Specify Parameters in the Platform Designer Component Editor...........................300
  5.9.1. Valid Ranges for Parameters in the _hw.tcl File............................................302
  5.9.2. Types of Platform Designer Parameters..........................................................303
  5.9.3. Declare Parameters with Custom _hw.tcl Commands........................................304
  5.9.4. Validate Parameter Values with a Validation Callback.....................................306
5.10. Declaring SystemVerilog Interfaces in _hw.tcl..................................................306
5.11. User Alterable HDL Parameters in _hw.tcl........................................................308
5.12. Scripting Wire-Level Expressions.......................................................................310
5.13. Control Interfaces Dynamically with an Elaboration Callback............................310
5.14. Control File Generation Dynamically with Parameters and a Fileset Callback..........311
5.15. Create a Composed Component or Subsystem....................................................312
5.16. Create an IP Component with Platform Designer a System View Different from
the Generated Synthesis Output Files....................................................................314
5.17. Add Component Instances to a Static or Generated Component..........................315
  5.17.1. Static Components.......................................................................................316
  5.17.2. Generated Components.............................................................................317
  5.17.3. Design Guidelines for Adding Component Instances....................................320
5.18. Creating Platform Designer Components Revision History........................................320
6. Platform Designer Command-Line Utilities..........................................................322
  6.1. Run the Platform Designer Editor with qsys-edit................................................322
  6.2. Scripting IP Core Generation.............................................................................324
    6.2.1. qsys-generate Command-Line Options.........................................................325
  6.3. Display Available IP Components with ip-catalog.............................................326
  6.4. Create an .ipx File with ip-make-ixp...............................................................327
  6.5. Generate Simulation Scripts..............................................................................328
  6.6. Generate a Platform Designer System with qsys-script.......................................329
  6.7. Platform Designer Scripting Command Reference..............................................331
    6.7.1. System........................................................................................................332
    6.7.2. Subsystems.................................................................................................345
    6.7.3. Instances....................................................................................................354
    6.7.4. Connections...............................................................................................387
    6.7.5. Top-level Exports.......................................................................................399
    6.7.6. Validation....................................................................................................412
    6.7.7. Miscellaneous............................................................................................418
    6.7.8. Wire-Level Connection Commands.............................................................431
  6.8. Platform Designer Scripting Property Reference.................................................435
    6.8.1. Connection Properties................................................................................436
    6.8.2. Design Environment Type Properties..........................................................437
    6.8.3. Direction Properties....................................................................................438
    6.8.4. Element Properties....................................................................................439
    6.8.5. Instance Properties....................................................................................440
    6.8.6. Interface Properties....................................................................................441
## Contents

6.8.7. Message Levels Properties................................................................. 442
6.8.8. Module Properties.............................................................................. 443
6.8.9. Parameter Properties........................................................................ 444
6.8.10. Parameter Status Properties............................................................ 446
6.8.11. Parameter Type Properties................................................................. 447
6.8.12. Port Properties.................................................................................. 448
6.8.13. Project Properties............................................................................... 449
6.8.14. System Info Type Properties.............................................................. 450
6.8.15. Units Properties................................................................................ 452
6.8.16. Validation Properties........................................................................ 453
6.8.17. Interface Direction............................................................................ 454
6.8.18. File Set Kind..................................................................................... 455
6.8.19. Access Type...................................................................................... 456
6.8.20. Instantiation HDL File Properties..................................................... 457
6.8.21. Instantiation Interface Duplicate Type.............................................. 458
6.8.22. Instantiation Interface Properties...................................................... 459
6.8.23. Instantiation Properties..................................................................... 460
6.8.25. VHDL Type....................................................................................... 462
6.9. Platform Designer Command-Line Interface Revision History............. 462

7. **Component Interface Tcl Reference**.................................................. 463

7.1. Platform Designer _hw.tcl Command Reference............................... 463
  7.1.1. Interfaces and Ports......................................................................... 464
  7.1.2. Parameters....................................................................................... 482
  7.1.3. Display Items.................................................................................. 493
  7.1.4. Module Definition............................................................................ 500
  7.1.5. Composition.................................................................................... 512
  7.1.6. Fileset Generation.......................................................................... 532
  7.1.7. Miscellaneous................................................................................ 543
  7.1.8. SystemVerilog Interface Commands.............................................. 549
  7.1.9. Wire-Level Expression Commands................................................ 555

7.2. Platform Designer _hw.tcl Property Reference.................................. 559
  7.2.1. Script Language Properties............................................................. 560
  7.2.2. Interface Properties........................................................................ 561
  7.2.3. SystemVerilog Interface Properties................................................ 561
  7.2.4. Instance Properties......................................................................... 563
  7.2.5. Parameter Properties........................................................................ 564
  7.2.6. Parameter Type Properties.............................................................. 566
  7.2.7. Parameter Status Properties............................................................ 567
  7.2.8. Port Properties................................................................................ 568
  7.2.9. Direction Properties........................................................................ 570
  7.2.10. Display Item Properties................................................................. 571
  7.2.11. Display Item Kind Properties......................................................... 572
  7.2.12. Display Hint Properties................................................................. 573
  7.2.13. Module Properties......................................................................... 574
  7.2.14. Fileset Properties.......................................................................... 576
  7.2.15. Fileset Kind Properties................................................................. 577
  7.2.16. Callback Properties....................................................................... 578
  7.2.17. File Attribute Properties............................................................... 579
  7.2.18. File Kind Properties..................................................................... 580
  7.2.19. File Source Properties................................................................. 581
1. Creating a System with Platform Designer

The Intel® Quartus® Prime software includes the Platform Designer system integration tool. Platform Designer simplifies the task of defining and integrating custom IP components (IP cores) into your FPGA design.

Platform Designer automatically creates interconnect logic from high-level connectivity that you specify. The interconnect automation eliminates the time-consuming task of specifying system-level HDL connections.

Platform Designer allows you to specify interface requirements and integrate IP components within a graphical representation of the system. The Intel Quartus Prime software installation includes the Intel FPGA IP library available from the IP Catalog in Platform Designer.

You can integrate optimized and verified Intel FPGA IP cores into a design to shorten design cycles and maximize performance. Platform Designer also supports integration of IP cores from third-parties, or custom components that you define.

Platform Designer provides support for the following:

- Create and reuse components—define and reuse custom parameterizable components in a Hardware Component Definition File (_hw.tcl) that describes and packages IP components.
- Command-line support—optionally use command-line utilities and scripts to perform functions available in the Platform Designer GUI.
- Up to 64-bit addressing.
- Optimization of interconnect and pipelining within the system and auto-adaptation of data widths and burst characteristics.
- Inter-operation between standard protocols.

Related Information
- Platform Designer Command-Line Utilities on page 322
- Introduction to Intel FPGA IP Cores
- Platform Designer System Design Flow on page 12

1.1. Platform Designer Interface Support

Platform Designer is most effective when you use standard interfaces available in the IP Catalog to design custom IP. Standard interfaces operate efficiently with Intel FPGA IP components, and you can take advantage of the bus functional models (BFMs), monitors, and other verification IP that the IP Catalog provides.
Platform Designer supports the following interface specifications:

- Intel FPGA Avalon® Memory-Mapped and Streaming
- Arm® AMBA® 3 AXI (version 1.0)
- Arm AMBA 4 AXI (version 2.0)
- Arm AMBA 4 AXI-Lite (version 2.0)
- Arm AMBA 4 AXI-Stream (version 1.0)
- Arm AMBA 3 APB (version 1.0)

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 Platform Designer system, or export outside of a Platform Designer system.

Platform Designer IP components can include the following 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 can be processors and DMAs, while slave memory devices can be RAMs, ROMs, and control registers. Data transfers between master and slave may be uni-directional (read only or write only), or bi-directional (read and write).</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. Platform Designer 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, Platform Designer 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 you export from the Platform Designer system. Platform Designer uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Platform Designer system as a point-to-point connection. Alternatively, you can export conduit interfaces and bring the interfaces 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 Platform Designer system.</td>
</tr>
</tbody>
</table>

### Related Information
- Avalon Interface Specifications
- AMBA Protocol Specifications
1.2. Platform Designer System Design Flow

You can use the Platform Designer GUI to quickly create and customize a Platform Designer system for integration with an Intel Quartus Prime project. Alternatively, you can perform many of the functions available in the Platform Designer GUI at the command-line, as Platform Designer Command-Line Utilities on page 322 describes.

When you create a system in the GUI, Platform Designer creates a .qsys or .qip file that represents the system in your Intel Quartus Prime software project.

The circled numbers in the diagram correspond with the following topics in this chapter:
1. Starting or Opening a Project in Platform Designer on page 12
2. Adding IP Components to a System on page 21
3. Connecting System Components on page 29
4. Specifying Interconnect Requirements on page 35
5. Synchronizing System Component Information on page 52
6. Generating a Platform Designer System on page 53
7. Simulating a Platform Designer System on page 61
8. Integrating a Platform Designer System with the Intel Quartus Prime Software on page 63

1.3. Starting or Opening a Project in Platform Designer

1. To start a new Platform Designer project, save the default system that appears when you open Platform Designer (File ➤ Save), or click File ➤ New System, and then save your new project. Platform Designer saves the new project in the Intel Quartus Prime project directory. To alternatively save your Platform Designer project in a different directory, click File ➤ Save As.

2. To open a recent Platform Designer 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 Platform Designer 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.

1.4. Viewing a Platform Designer System

Platform Designer allows you to visualize all aspects of your system. By default, Platform Designer displays the contents of your system in the System Contents whenever you open a system. You can also access other panels that allow you to view and modify various elements of the system.

When you select or edit an item in one Platform Designer tab, all other tabs update to reflect your selection or edit. For example, if you select the cpu_0 in the Hierarchy tab, the Parameters tab immediately updates to display cpu_0 parameters.

Click the View menu to interact with the elements of your system in various tabs.
The Platform Designer GUI is fully customizable. You can arrange and display Platform Designer GUI elements that you most commonly use, and then save and reuse useful GUI layouts.

The IP Catalog and Hierarchy tabs display to the left of the main frame by default. 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 Platform Designer. 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 Platform Designer window.

**Figure 1. View a Platform Designer System**

![](image)

1.4.1. Viewing the System Hierarchy

The Hierarchy tab hierarchically displays the modules, connections, and exported signals in the current system. You can expand and traverse though the system hierarchy, zoom in for detail, and locate to elements in other Platform Designer panes.

The Hierarchy tab provides the following information and functionality:

- Lists connections between components.
- Lists names of signals in exported interfaces.
- Right-click to connect, edit, add, remove, or duplicate elements in the hierarchy.
- Displays internal connections of Platform Designer subsystems that you include as IP components. By contrast, the System Contents tab displays only the exported interfaces of Platform Designer subsystems.
Expanding the System Hierarchy

Click the + icon to expand any interface in the Hierarchy tab to view sub-components, associated elements, and signals for the interface. The Hierarchy tab displays a unique icon for each element type in the system. In the example below, the ram_master selection appears selected in both the System Contents and Hierarchy tabs.

1.4.2. Filtering the System Contents

You can easily filter the display of your system in the System Contents by interface type, instance name, or other custom properties that you define. Filtering the view allows you to simplify the display and focus only on the items you want.

For example, you can click the Filter button to display only instances that include memory-mapped interfaces, or display only instances that connect to a particular Nios® II processor. Conversely, you can temporarily hide clock and reset interfaces to further simplify the display.

Figure 2. Filter Icon and Filters Dialog Box

Related Information

Filters Dialog Box
1.4.3. Viewing Clock and Reset Domains

The Platform Designer **Clock Domains** and **Reset Domains** tabs list the clock and reset domains in the Platform Designer system, respectively.

Click **View ➤ Clock Domains** or click **View ➤ Reset Domains** to display these tabs.

Platform Designer determines clock and reset domains by the associated clocks and resets. This information displays when you hover over interfaces in your system.

The **Clock Domains** and **Reset Domains** tabs also allow you to locate system performance bottlenecks. The tabs indicate connection points where Platform Designer automatically inserts clock-crossing adapters and reset synchronizers during system generation. View the following information on these tabs to create optimal connections between interfaces:

- The number of clock and reset domains in the system
- The interfaces and modules that each clock or reset domain contains
- The locations of clock or reset crossings
- The connection point of automatically inserted clock or reset adapters
- The proper location for manual insertion of a clock or reset adapter

**Figure 3.** Platform Designer Clock and Reset Domains

1.4.3.1. Viewing Clock Domains in a System

On 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 the **Filter** control. When you select an element in the **Clock Domains** tab, the corresponding selection appears highlighted in the **System Contents** tab.

Follow these steps to filter and highlight clock domains in the **System Contents**:

1. Click **View ➤ Clock Domains**.
2. Select any clock or reset domain in the list to view associated interfaces. The corresponding selection appears in the **System Contents** tab.
3. To highlight clock domains in the **System Contents** tab, click **Show clock domains in the system table** or at the bottom of the **System Contents** tab.
4. To view a single clock domain, or multiple clock domains and their modules and connections, select the clock name or names in the **Clock Domains** tab. The modules for the selected clock domain or domains and connections highlight in the **System Contents** tab. Detailed information for the current selection appears in the clock domain details pane.

**Figure 5. Clock Domains**

![Clock Domains](image)

*Note:* If a connection crosses a clock domain, the connection circle appears as a red dot in the **System Contents** tab.

5. 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.

Platform Designer lists the interfaces that cross clock domains under **Clock Domain Crossings**. As you click through the elements, detailed information appears in the clock domain details pane. Platform Designer also highlights the selection in the **System Contents** tab.

### 1.4.3.2. Viewing Reset Domains in a System

On 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.

Follow these steps to filter and highlight reset domains in the **System Contents**:

1. To open the **Reset Domains** tab, click **View ➤ Reset Domains**.
2. To show reset domains in the **System Contents** tab, click the **Show reset domains in the system table** icon in the **System Contents** tab.
3. To view a single reset domain, or multiple reset domains and their modules and connections, click the reset names in the **Reset Domain** tab.

Platform Designer 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 domains 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 Platform Designer during system generation, for example, a reset synchronizer. Platform Designer 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.

### 1.4.4. Viewing Avalon Memory-Mapped Domains in a System

The **Avalon Memory Mapped Domains** tab displays a list of all the Avalon domains in the system. When you select a domain in the **Avalon Memory Mapped Domains** tab, the corresponding selection highlights in the **System Contents** tab.

Click **View ➤ Avalon Memory Mapped Domains** to display this tab.
Filter the **System Contents** tab to display a single Avalon domain, or multiple domains. Further filter your view with selections in the **Filters** dialog box.

To rename an Avalon memory-mapped domain, double-click the domain name. Detailed information for the current selection appears in the Avalon domain details pane.

To enable and disable the highlighting of the Avalon domains in the **System Contents** tab, click the domain control tool at the bottom of the **System Contents** tab.
1.4.5. Viewing the System Schematic

The Schematic tab displays a schematic representation of the current Platform Designer system. You can zoom into a component or connection to view more details. You can use the image handles in the right panel to resize the schematic image.

If your selection is a subsystem, you can use the `Move to the top of the hierarchy`, `Move up one level of hierarchy`, and `Drill into a subsystem to explore its contents` buttons to traverse the schematic of a hierarchical system.

Figure 9. Platform Designer Schematic Tab

![Platform Designer Schematic Tab](image)

Related Information

Editing a Subsystem on page 67

1.4.6. Viewing System Assignments and Connections

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 list of connections in your Platform Designer 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.
1.4.7. Customizing the Platform Designer Layout

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.

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. 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, Platform Designer also saves the current workspace configuration. When you re-open a saved system, Platform Designer restores the last saved workspace.

The **Reset to System Layout** command on the View menu restores the workspace to its default configuration for Platform Designer system design. The **Reset to IP Layout** command restores the workspace to its default configuration for defining and generating single IP cores.

Follow these steps to customize and save the Platform Designer layout:

1. Click items on the View menu to display and then optionally dock the tabs. Rearrange the tabs to suit your preferences.

2. To save the current Platform Designer window configuration as a custom layout, click **View ➤ Custom Layouts ➤ Save**. Platform Designer 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 determines the order of layouts in the list.

3. Use any of the following methods to revert to another layout:
• To revert the layout to the default system design layout, click View ➤ Reset to System Layout. This layout displays the System Contents, Address Map, Interconnect Requirements, and Messages tabs in the main pane, and the IP Catalog and Hierarchy tabs along the left pane.

• To revert the layout to the default system design layout, click View ➤ Reset to IP Layout. This layout displays the Parameters and Messages tabs in the main pane, and the Details, Block Symbol, and Presets tabs along the right pane.

• To reset your Platform Designer window configuration to a previously saved layout, click View ➤ Custom Layouts, and then select the custom layout.

• Press Ctrl+3 to quickly change the Platform Designer layout.

4. 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.

Figure 11. Manage Custom Layouts

1.5. Adding IP Components to a System

You can quickly add Intel FPGA IP components to a system from the IP Catalog in Platform Designer. The IP Catalog launches a parameter editor that allows you to specify options and add the component to your system. Your Platform Designer system can contain a single instance of an IP component, or multiple, individually parameterized variations of multiple or the same IP components.

Follow these steps to locate, parameterize, and instantiate an IP component in a Platform Designer system:

1. To locate a component by name, 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. You can also find components by category.
2. Double-click any component to launch the component's parameter editor and specify options for the component.

For some IP components, you can select and Apply a pre-defined set of parameters values for specific applications from the Presets list.

3. To complete customization of the IP component, click Finish. The IP component appears in the System Contents tab.
1. Creating a System with Platform Designer

1.5.1. Modifying IP Parameters

The **Parameters** tab allows you to view and edit the current parameter settings for IP components in your system.

To display a component’s parameters on the **Parameters** tab:

1. Click **View ➤ Parameters**.
2. Select the component in the **System Contents** or **Hierarchy** tabs.

The **Parameters** tab provides the following functionality:

- **Parameters** field—adjust the parameters to align with your design requirements, including changing the name of the top-level instance.
- Component Banner—displays the hierarchical path for the component and allows you to enable display of internal names. Below the hierarchical path, the parameter editor shows the HDL entity name and the IP file path for the selected IP component. Right-click in the banner to display internal parameter names for use with scripted flows.
- **Read/Write Waveforms**—displays the interface timing and the corresponding read and write waveforms.
- **Details**—displays links to detailed information about the component.
- **Parameterization Messages**—displays parameter warning and error messages about the IP component.

**Figure 14. Platform Designer Parameters Tab**
Changes that you make in the Parameters tab affect your entire system, and dynamically update other open tabs in Platform Designer. Any change that you make on the Parameters tab, automatically updates the corresponding .ip file that stores the component's parameterization.

If you create your own custom IP components, you can use the Hardware Component Description File (.hw.tcl) to specify configurable parameters.

**Note:**

If you use the ip-deploy or qsys-script commands rather than the Platform Designer GUI, you must use internal parameter names with these parameters.

### 1.5.1.1. Viewing Component or Parameter Details

The Details tab provides information for a component or parameter that you select. Platform Designer updates the information in the Details tab as you select different components.

To view a component's details:

1. Click the parameters for a component in the parameter editor, Platform Designer displays the description of the parameter in the Details tab.
2. To return to the complete description for the component, click the header in the Parameters tab.

### 1.5.1.2. Viewing a Component's Block Symbol

The Block Symbol tab displays a symbolic representation of any component you select in the Hierarchy or System Contents tabs. The block symbol shows the component's port interfaces and signals. The Show signals option allows you to turn on or off signal graphics.

**Figure 15. Block Symbol Tab**

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.
1.5.2. Applying Preset Parameters for Specific Applications

The **Preset** tab displays the names of any available preset settings for an IP component. The preset preserves a collection of parameter settings that may be appropriate for a specific protocol or application. Not all IP components include preset parameters. Double-click the preset parameter name to apply the preset parameter values to a component you are defining.

**Figure 16. Selecting Preset Parameters**

---

1.5.2.1. Creating IP Custom Preset Parameters Settings

You can optionally define and save a custom set of parameter settings for an IP component, and then apply the preset settings whenever you add an instance of the IP component to any system.

Follow these steps to save custom preset parameter settings:

1. In IP Catalog, double-click any component to launch the parameter editor.
2. To search for a specific preset for the initial settings, type a partial preset name in the search box.
3. In the **Presets** tab, click **New** to specify the **Preset name** and **Preset description**.
4. Under **Select parameters to include in the preset**, enable or disable the parameters you want to include in the preset.
5. Specify the path for the **Preset file** that preserves the collection of parameter settings.
Figure 17. Create New Preset

If the file location that you specify is not already in the IP search path, Platform Designer adds the location of the new preset file to the IP search path.

6. Click **Save**.
7. To apply the preset to an IP component, click **Apply**. Preset parameter values that match the current parameter settings appear in bold.

### 1.5.3. Adding Third-Party IP Components

You can add third-party IP components created by Intel partners to your Platform Designer system. Third-party partner IP components have interfaces that Platform Designer supports, such as Avalon-MM or AMBA AXI. Third-party partner IP components can also include timing and placement constraints, software drivers, simulation models, and reference designs.

To locate supported third-party IP components on the Intel web page, follow these steps:

1. From the Intel website, navigate to the **Find IP** page, and then click Find IP on the tool.
2. Use the **Search** box and the **End Market**, **Technology**, **Devices** or **Provider** filters to locate the IP that you want to use.
3. Click **Enter**.
4. Sort the table of results for the **Platform Designer Compliant** column. You cannot use non-compliant components in Platform Designer.
5. Click the IP name to view information, request evaluation, or request download.
6. After you download the IP files, add the IP location to the IP search path to add the IP to IP Catalog, as **IP Search Path Recursive Search** on page 27 describes.

**Related Information**

Find Intel FPGA and Partner IP
1.5.3.1. IP Search Path Recursive Search

The Intel Quartus Prime software automatically searches and identifies IP components in the IP search path. The search is recursive for some directories, and only to a specific depth for others. You can use ** characters to halt a recursive search at any directory that contains a _hw.tcl or .ipx file.

In the following list of search locations, ** indicates a recursive descent.

<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 Intel 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 Intel Quartus Prime project directory.</td>
</tr>
</tbody>
</table>

1.5.3.1.1. IP Search Path Precedence

If the Intel 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.

1.5.3.1.2. IP Component Description Files

The Intel Quartus Prime software identifies parameterizable IP components in the IP search path for the following 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. This file specifies the relative path of directories to search for IP cores. In general, .ipx files facilitate faster searches.

1.5.3.2. Defining the IP Search Path with Index Files

You can create an IP Index File (.ipx) to specify a path that Platform Designer searches for IP components.

You can specify a search path in the user_components.ipx file in either in Platform Designer (Tools ➤ Options) or the Intel 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 Platform Designer to the location of an IP component or directory to search.
A `<path>` element in a .ipx file specifies a directory where Platform Designer can search for IP components. A `<component>` entry specifies the path to a single component. `<path>` elements allow wildcards in definitions. An asterisk matches any file name. If you use an asterisk as a directory name, it matches any number of subdirectories.

**Example 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 Platform Designer is less than if Platform Designer must discover the files in a directory.

**Example 2. Component Element in an .ipx File**

The example shows two `<component>` elements. Note that the paths for file names are specified relative to the .ipx file.

```xml
<library>
  <component
      name="A Platform Designer Component"
      displayName="Platform Designer 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 327

### 1.5.4. Creating or Opening an IP Core Variant

In addition to creating a system, Platform Designer allows you to define a stand-alone IP core variant that you can add to your Intel Quartus Prime project or to a Platform Designer system.

Follow these steps to define an IP core variant in Platform Designer:

1. In Platform Designer, click **File ➤ New IP Variant.**
2. On the **IP Variant** tab, specify the **Quartus project** to contain the IP variant.
3. Specify any of the following options:
   - **Revision**—optionally select a specific revision of a project.
   - **Device family**—when defining a new project or None, allows you to specify the target Intel FPGA device family. Otherwise this field is non-editable and displays the Quartus project target device family. Click Retrieve Values to populate the fields.
   - **Device part**—when defining a new project or None, allows you to specify the target Intel FPGA device part number. Otherwise this field is non-editable and displays the Quartus project target device part number.

4. Specify the **IP variant** name, or browse for an existing IP variant.

5. For **Component type**, click Select and select the IP component from the IP Catalog.

6. Click Create. The IP parameter editor appears. Specify the parameter values that you want for the IP variant.

7. To generate the IP variant synthesis and optional simulation files, click **Generate HDL**, specify **Generation Options**, and click Generate. Refer to **Generation Dialog Box Options** on page 54 for generation options.

### 1.6. Connecting System Components

You must appropriately connect the components in your Platform Designer system. The **System Contents** and **Connections** tabs allow you to connect and configure IP components quickly. Platform Designer supports connections between 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 Platform Designer system within a parent system.
Platform Designer uses the high-level connectivity you specify to instantiate a suitable HDL fabric to perform the needed adaptation and arbitration between components. Platform Designer generates and includes this interconnect fabric in the RTL system output.

Potential connections between interfaces appear as gray interconnect lines with an open circle icon at the intersection of the potential connection.

**Figure 19. Potential and Implemented Connections in System Contents**

<table>
<thead>
<tr>
<th>Connections</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock_in</td>
</tr>
<tr>
<td></td>
<td>in_clk</td>
</tr>
<tr>
<td></td>
<td>out_clk</td>
</tr>
<tr>
<td></td>
<td>reset_in</td>
</tr>
<tr>
<td></td>
<td>clk</td>
</tr>
<tr>
<td></td>
<td>in_reset</td>
</tr>
<tr>
<td></td>
<td>out_reset</td>
</tr>
<tr>
<td></td>
<td>master_0</td>
</tr>
<tr>
<td></td>
<td>clk</td>
</tr>
<tr>
<td></td>
<td>clk_reset</td>
</tr>
<tr>
<td></td>
<td>master</td>
</tr>
<tr>
<td></td>
<td>master_reset</td>
</tr>
</tbody>
</table>

To implement a connection, follow these steps:

1. Click inside an open connection circle to implement the connection between the interfaces. When you make a connection, Platform Designer changes the connection line to black, and fills the connection circle. Clicking a filled-in circle removes the connection.

2. To display the list of current and possible connections for interfaces in the Hierarchy or System Contents tabs, click View ➤ Connections.

**Figure 20. Connection Display for Exported Interfaces**

3. Perform any of the following to modify connections:
• On the Connections tab, enable or disable the Connected column to enable or disable any connection. The Clock Crossing, Data Width, and Burst columns provide interconnect information about added adapters that can result in slower f<sub>MAX</sub> or increased area utilization.

• On the System Contents tab, right-click in the Connection column and disable or enable Allow Connection Editing.

• On the Connections tab view and make connections for exported interfaces. Double-click an interface in the Export column to view all possible connections in the Connections column as pins. To restore the representation of the connections, and remove the interface from the Export column, click the pin.

1.6.1. Platform Designer 64-Bit Addressing Support

Platform Designer interconnect supports up to 64-bit addressing for all Platform Designer interfaces and IP components, with a range of: 0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF, inclusive.

The 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. Platform Designer 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 Platform Designer 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.

Intel Quartus Prime debugging tools provide access to the state of an addressable system via the Avalon-MM interconnect. These tools are also 64-bit compatible, and process within a 64-bit address space, including a JTAG to Avalon master bridge.

Platform Designer supports auto base address assignment for Avalon-MM components. In the Address Map tab, click Auto Assign Base Address.

Related Information
• Address Map Tab Help
• Address Span Extender on page 228
• auto_assign_base_addresses on page 419

1.6.1.1. Support for Avalon-MM Non-Power of Two Data Widths

Platform Designer requires that you connect all multi point Avalon-MM connections to interfaces with data widths that are equal to powers of two.

Platform Designer 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, you can set their data widths to a non-power of two.
1.6.2. Connecting Masters and Slaves

Specify connections between master and slave components in the **Address Map** tab. This tab allows you to specify the address range that each memory-mapped master uses to connect to a slave in a Platform Designer system.

The **Address Map** tab 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. In this case, use the **Address Map** tab to view the individual memory addresses for each connected master.

Platform Designer enables you to design a system where two masters access the same slave at different addresses. If you use this feature, Platform Designer labels the **Base** and **End** address columns in the **System Contents** tab as "mixed" rather than providing the address range.

To create or edit a connection between master and slave IP components:

1. In Platform Designer, click 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. 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 Platform Designer interconnect, which provides an efficient address decoding logic, which in turn allows Platform Designer to achieve the best possible $f_{\text{MAX}}$.

![Figure 21. Address Map Tab for Connection Masters and Slaves](image)

**Assigns Base Address**

**Slave to Master Address Mapping**

**Related Information**

- Address Map Tab Help
- Platform Designer 64-Bit Addressing Support on page 31
- auto_assign_base_addresses on page 419
1.6.3. Changing a Conduit to a Reset

1. In the IP Catalog search box, locate **IOPLL Intel FPGA IP** and double-click to add the component to your system.

2. In the **System Contents** tab, select the **PLL** component.

3. Click **View ➤ Component Instantiation** and open the **Component Instantiation** tab for the selected component.

4. In the **Signals & Interfaces** tab, select the **locked conduit interface**.

5. Change the **Type** from **Conduit** to **Reset Input**, and the **Synchronous edges** from **Deassert** to **None**.

6. Select the **locked [1]** signal below the **locked interface**.

7. Change the **Signal Type** from **export** to **reset_n**. Change the **Direction** from **output** to **input**.

8. Click **Apply**.

The conduit interface changes to reset for the instantiated PLL component.

Figure 22. **Changing Conduit to a Reset**

1.6.4. Previewing the System Interconnect

You can review a graphical representation of the Platform Designer interconnect before you generate the system. The **System with Platform Designer Interconnect** window shows how Platform Designer converts connections between interfaces to interconnect logic during system generation.

To open the **System with Platform Designer Interconnect** window, click **System ➤ Show System With Platform Designer Interconnect**.
The System with Platform Designer 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.

**Figure 23. System with Platform Designer Interconnect window**

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 Platform Designer Interconnect window displays a graphical representation of command and response datapaths in your system. Datapaths allow you precise control over pipelining in the interconnect. Platform Designer displays separate figures for the command and response datapaths. You can access the datapaths 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 datapaths between modules. Turn on **Show Pipelinable 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.

1.7. Specifying Interconnect Requirements

The Interconnect Requirements tab allows you to apply system-wide ($system) or interface-specific interconnect requirements for IP components in your system.

Available options in the Setting column vary, depending on the Identifier column value. Click the drop-down menu to select the settings, and to assign the corresponding values to the settings.

Follow these steps to specify system or interface interconnect requirements.

1. To create a new Identifier to assign an interconnect requirement, click Add. A new_target row appears for edit.

2. Click the new_target cell and select $system to define a system-wide requirement, or select any interface name to specify interconnect requirements for the interface.

3. In the same row, click the new_requirement cell, select any of the available requirements, as Interconnect Requirements on page 35 describes.

4. In the same row, Click the new_requirement_value cell and specify the requirement value.


Related Information
• Platform Designer Interconnect on page 124
• Reset Interfaces on page 167

1.7.1. Interconnect Requirements

Table 3. System-Wide Interconnect Requirements

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit interconnect pipeline stages to</td>
<td>Specifies the maximum number of pipeline stages that Platform Designer can insert in each command and response path to increase the f_MAX at the expense of additional latency. You can specify between 0 and 4 pipeline stages, where 0 means that the interconnect has a combinational datapath. This setting is specific for each Platform Designer system or subsystem.</td>
</tr>
<tr>
<td>Clock crossing adapter type</td>
<td>Specifies the default implementation for automatically inserted clock crossing adapters:</td>
</tr>
<tr>
<td>Handshake</td>
<td>This adapter uses a simple handshaking 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.</td>
</tr>
<tr>
<td>FIFO</td>
<td>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 continued...</td>
</tr>
</tbody>
</table>


### Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>require more resources. The <strong>FIFO</strong> adapter is appropriate for memory-mapped transfers requiring high throughput across clock domains.</td>
<td>If you select <strong>Auto</strong>, Platform Designer specifies the <strong>FIFO</strong> adapter for bursting links, and the <strong>Handshake</strong> adapter for all other links.</td>
</tr>
<tr>
<td><strong>Auto</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Automate default slave insertion**

Directs Platform Designer to automatically insert a default slave for undefined memory region accesses during system generation.

**Enable instrumentation**

When you set this option to **TRUE**, Platform Designer enables debug instrumentation in the Platform Designer interconnect, which then monitors interconnect performance in the system console.

**Burst Adapter Implementation**

Allows you to choose the converter type that Platform Designer applies to each burst.

<table>
<thead>
<tr>
<th>Converter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic converter</strong> <em>(slower, lower area)</em></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Per-burst-type converter</strong> <em>(faster, higher area)</em></td>
<td>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 $f_{\text{MAX}}$.</td>
</tr>
</tbody>
</table>

**Enable ECC protection**

Specifies the default implementation for ECC protection for memory elements.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FALSE</strong></td>
<td>Default. Disables ECC protection for memory elements in the Platform Designer interconnect.</td>
</tr>
<tr>
<td><strong>TRUE</strong></td>
<td>Enables ECC protection for memory elements. Platform Designer interconnect sends uncorrectable errors arising from memory as $\text{DECODEERROR (DECERR)}$ on the Avalon response bus.</td>
</tr>
</tbody>
</table>

For more information about Error Correction Coding (ECC), refer to [Error Correction Coding (ECC) in Platform Designer Interconnect](on page 184).

### Table 4. Specifying Interface Interconnect Requirements

You can apply the following interconnect requirements when you select a component interface as the **Identifier** in the **Interconnect Requirements** tab, in the **All Requirements** table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Security</strong></td>
<td>• Non-secure&lt;br&gt;• Secure&lt;br&gt;• Secure ranges&lt;br&gt;• TrustZone*-aware</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.</td>
</tr>
<tr>
<td><strong>Secure address ranges</strong></td>
<td>Accepts valid address range.</td>
<td>Allows you to type in any valid address range.</td>
</tr>
</tbody>
</table>

**Related Information**

- Interconnect Pipelining on page 180
- Error Correction Coding (ECC) in Platform Designer Interconnect on page 184
1.8. Defining Instance Parameters

You can implement instance parameters to test the functionality of your Platform Designer system when using another Platform Designer system as a sub-component of a larger system. A higher-level Platform Designer system can assign values to instance parameters, and then test those values in the lower-level system.

You can use the Instance Parameters tab to define how the specified values for the instance parameters affect the sub-components in the Platform Designer system. You define an Instance Script that creates queries for the instance parameters, and sets the values of the parameters for the lower-level system components.

When you click Preview Instance, Platform Designer creates a preview of the current Platform Designer 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 Platform Designer system must have parameters that can be set when they are instantiated in a higher-level system.

If you create hierarchical Platform Designer systems, each Platform Designer system in the hierarchy can include instance parameters to pass parameter values through multiple levels of hierarchy.

1.8.1. Creating an Instance Parameter Script in Platform Designer

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 Intel 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 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 Platform Designer 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 463
1.8.2. Platform Designer Instance Parameter Script Tcl Commands

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 Platform Designer system, or to set the values of the parameters of the sub-IP-components instantiated in the system.

1.8.2.1. get_instance_parameter_value

**Description**
Returns the parameter value in a child instance.

**Usage**
```
get_instance_parameter_value <instance> <parameter>
```

**Returns**
`various` The parameter value.

**Arguments**

`instance` The instance name.

`parameter` The parameter name.

**Example**
```
get_instance_parameter_value pixel_converter input_DPI
```

**Related Information**
- `get_instance_parameters` on page 40
- `set_instance_parameter_value` on page 384
1.8.2.2. get_instance_parameters

Description
Returns the names of all parameters for a child instance that the parent can manipulate. This command omits derived parameters and parameters that have the SYSTEM_INFO parameter property set.

Usage
get_instance_parameters <instance>

Returns
instance  The list of parameters in the instance.

Arguments
instance  The instance name.

Example
get_instance_parameters uart_0

Related Information
- get_instance_parameter_property on page 374
- get_instance_parameter_value on page 39
- set_instance_parameter_value on page 384
1.8.2.3. get_parameter_value

**Description**
Returns the current value of a parameter defined previously with the `add_parameter` command.

**Usage**
`get_parameter_value <parameter>`

**Returns**
The value of the parameter.

**Arguments**

*parameter*  The name of the parameter whose value is being retrieved.

**Example**

```
get_parameter_value fifo_width
```
1.8.2.4. 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
### 1.8.2.5. send_message

**Description**

Sends a message to the user of the component. The message text is normally HTML. You can use the `<b>` element to provide emphasis. If you do not want the message text to be HTML, then pass a list like `{ Info Text }` as the message level.

**Usage**

```
send_message <level> <message>
```

**Returns**

No return value.

**Arguments**

- **level** Intel Quartus Prime supports the following message levels:
  - `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!"
```
1.8.2.6. 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**

```plaintext
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**

```plaintext
set_instance_parameter_value uart_0 baudRate 9600
```
1.8.2.7. set_module_property

**Description**
Specifies the Tcl procedure to evaluate changes in Platform Designer system instance parameters.

**Usage**

```
set_module_property <property> <value>
```

**Returns**
No return value.

**Arguments**

- `property` The property name. Refer to Module Properties.
- `value` The new value of the property.

**Example**

```
set_module_property COMPOSITION_CALLBACK "my_composition_callback"
```

**Related Information**
- `get_module_properties` on page 337
- `get_module_property` on page 338
- Module Properties on page 443

1.9. Implementing Performance Monitoring

You can set up real-time performance monitoring for your Platform Designer system using throughput metrics such as read and write transfers.

Platform Designer supports performance monitoring for only Avalon-MM interfaces. In your Platform Designer system, you can monitor the performance of no less than three, and no greater than 15 Avalon-MM interface components at one time.

Follow these steps to implement performance monitoring:

1. Open a system in Platform Designer.
2. Click **View ➤ Instrumentation**.
3. To enable performance monitoring, turn on Add debug instrumentation to the Platform Designer Interconnect option. Enabling this option allows the system to interact with the Bus Analyzer Toolkit, accessible from the Intel Quartus Prime Tools menu.
4. For any interconnect, enable or disable the Add Performance Monitor option.
1.10. Configuring Platform Designer System Security

You can specify Platform Designer system and interconnect security settings on the Interconnect Requirements tab.

Platform Designer interconnect supports the Arm TrustZone security extension. The Platform Designer Arm TrustZone security extension includes secure and non-secure transaction designations, and a protocol for processing between the designations, as Table 6 on page 49 describes.

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.

AXI masters and slaves can be TrustZone-aware. All other master and slave interfaces, such as Avalon-MM interfaces, are non-TrustZone-aware.

The Avalon specification does not include a protection signal. Consequently, when an Avalon master sends a command, there is no embedded security and Platform Designer recognizes the command as non-secure. Similarly, when an Avalon slave receives a command, the slave always accepts the command and responds.

Follow these steps to set compile-time security support for non-TrustZone-aware components:

1. To begin creating a secure system, add masters and slaves to your system, as Adding IP Components to a System on page 21 describes.
2. Make connections between the masters and slaves in your system, as Connecting Masters and Slaves on page 32 describes.
3. Click View ➤ Interconnect Requirements. The Interconnect Requirements tab allows you to specify system-wide and interconnect-specific requirements.
4. To specify security requirements for an interconnect, click the Add button.
5. In the Identifier column, select the interconnect in the new_target cell.
6. In the **Setting** column, select **Security**.

7. In the **Value** column, select the appropriate **Secure**, **Non-Secure**, **Secure Ranges**, or **TrustZone-aware** security for the interface. Refer to System Security Options on page 48 for details of each option.

**Figure 25. Security Settings in Interconnect Requirements Tab**

8. 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, as Specifying a Default Slave on page 48.

**Related Information**

- Platform Designer Interconnect on page 124
- Platform Designer System Design Components on page 210
1.10.1. System Security Options

Table 5. Security Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure</td>
<td>Master sends only secure transactions, and the slave receives only secure transactions. Platform Designer treats transactions from a secure master as secure. Platform Designer blocks non-secure transactions to a secure slave and routes to the default slave.</td>
</tr>
<tr>
<td>Non-Secure</td>
<td>The master sends only non-secure transactions, and the slave receives any transaction, secure or non-secure. Platform Designer treats transactions from a non-secure master as non-secure. Platform Designer allows all transactions, regardless of security status, to reach a non-secure slave.</td>
</tr>
<tr>
<td>Secure Ranges</td>
<td>Applies to only the slave interface. Allows you to specify secure memory regions for a slave. Platform Designer blocks non-secure transactions to secure regions and routes to the default slave. 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>
| TrustZone-aware | TrustZone-aware masters have signals that control the security status of their transactions. TrustZone-aware slaves can accept these signals and handle security independently. The following applies to secure systems that mix secure and non-TrustZone-aware components:  
  - All AXI, AMBA 3 AXI, and AMBA 3 AXI-Lite masters are TrustZone-aware.  
  - You can set AXI, AMBA 3 AXI, and AMBA 3 AXI-Lite slaves as TrustZone-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. |

1.10.2. Specifying a Default Slave

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.

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 that includes a non-secure master that initiates transactions to a secure slave resulting in unsuccessful transfers, within the same hierarchy.

A transaction that violates security is rerouted to the default slave and subsequently responds to the master with an error. The following rules apply to specifying a default slave:

- You can designate any slave as the default slave.
- You can share a default slave between multiple masters.
- 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_error_response_slave` component includes the required TrustZone features.

To designate a slave interface as the default slave for non TrustZone-aware interfaces, follow these steps:

2. In the System Contents, right-click any column and turn on the Security and Default Slave columns.

3. In the System Contents tab, turn on the Default Slave option for the slave interface. A master can have only one default slave.

### Table 6. 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>
<tr>
<td>Non-TrustZone-aware memory (non-secure region)</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

**Related Information**

- Error Response Slave on page 233
- Designating a Default Slave on page 238

### 1.10.3. Accessing Undefined Memory Regions

Access to an undefined memory region occurs when a transaction from a master targets a memory region unspecified in the slave memory map. To ensure predictable response behavior when this condition occurs, you must specify a default slave, as Specifying a Default Slave on page 48 describes.

You can designate any memory-mapped slave as a default slave. Have only one default slave for each interconnect domain in your system. Platform Designer then routes undefined memory region accesses to the default slave, which terminates the transaction with an error response.

**Note:** If you do not specify the default slave, Platform Designer automatically assigns the slave at the lowest address within the memory map for the master that issues the request as the default slave.
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.

### 1.11. Upgrading Outdated IP Components

When you open a Platform Designer system that contains outdated IP components, Platform Designer automatically attempts to upgrade the IP components if it cannot locate the requested version.

Most Platform Designer IP components support automatic upgrade.

Platform Designer allows you to include a path to older IP components in the IP Search Path, and then use those components even if upgraded versions are available. However, older versions of IP components may not work in newer version of Platform Designer.

If a Platform Designer system includes IP components outside of the project directory or the directory of the .qsys file, you must add the location of these components to the Platform Designer IP Search Path (Tools ➤ Options).

To upgrade IP cores:

1. With the Platform Designer system open, click System ➤ Upgrade IP Cores. Only IP Components that are associated with the open Platform Designer system, and that do not support automatic upgrade appear in Upgrade IP Cores dialog box.
2. In the Upgrade IP Cores dialog box, select one or multiple IP components, and then click Upgrade. A green check mark appears for the IP components that Platform Designer successfully upgrades.
3. Generate the Platform Designer system.

Alternatively, you can upgrade IP components with the following command:

```bash
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. Platform Designer reports the start and finish of the command-line upgrade, but does not name the particular IP components upgraded.

For device migration information, refer to Introduction to Intel FPGA IP.

**Related Information**

Introduction to Intel FPGA IP Cores
1.11.1. Troubleshooting IP or Platform Designer System Upgrade

The Upgrade IP Components dialog box reports the version and status of each IP core and Platform Designer 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: Do not use spaces in IP variation names or paths.

During automatic or manual upgrade, the Messages window dynamically displays upgrade information for each IP core or Platform Designer system. Use the following information to resolve upgrade errors:

Table 7. IP Upgrade Error Information

<table>
<thead>
<tr>
<th>Upgrade IP Components Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Displays the &quot;Success&quot; or &quot;Failed&quot; status of each upgrade or migration. Click the status of any upgrade that fails to open the IP Upgrade Report.</td>
</tr>
<tr>
<td>Version</td>
<td>Dynamically updates the version number when upgrade is successful. The text is red when the IP requires upgrade.</td>
</tr>
<tr>
<td>Device Family</td>
<td>Dynamically updates to the new device family when migration is successful. The text is red when the IP core requires upgrade.</td>
</tr>
<tr>
<td>Auto 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 Platform Designer systems that fail upgrade. Review these reports to determine any port differences between the current and previous IP core version.</td>
</tr>
</tbody>
</table>

Use the following techniques to resolve errors if your IP core or Platform Designer 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:

- If the current version of the software does not support the IP variant, right-click the component and click Remove IP Component from Project. Replace this IP core or Platform Designer system with the one supported in the current version of the software.
- If the current target device does not support the IP variant, select a supported device family for the project, or replace the IP variant with a suitable replacement that supports your target device.
- If an upgrade or migration fails, click Failed in the Status field to display and review details of the IP Upgrade Report. Click the Release Notes link for the latest known issues about the IP core. Use this information to determine the nature of the upgrade or migration failure and make corrections before upgrade.
- Run Auto Upgrade to automatically generate an IP Ports Diff report for each IP core or Platform Designer system that fails upgrade. Review the reports to determine any port differences between the current and previous IP core version. Click Upgrade in Editor to make specific port changes and regenerate your IP core or Platform Designer system.
- If your IP core or Platform Designer system does not support Auto Upgrade, click Upgrade in Editor to resolve errors and regenerate the component in the parameter editor.
1.12. Synchronizing System Component Information

When a component instantiation values do match the component's corresponding .ip file, Platform Designer reports these mismatches as Component Instantiation Warnings in the System Messages tab.

You must synchronize any mismatches between the component instantiation, and the component's corresponding .ip prior to system generation.

Follow these steps to synchronize one or more components in your system:

1. Select the mismatched signal or interface in the System Contents tab, and then click View ➤ System Info. Alternatively, you can double-click the corresponding Component Instantiation Warning in the System Messages tab.
Figure 27. System Info Tab

2. View any component mismatches in the System Info tab. Select individual interfaces, signals, or parameters to view the specific value differences in the Component and IP file columns. Value mismatches between the Component Instantiation and the IP file appear in blue. Missing elements appear in green.

3. To synchronize the Component Instantiation and IP file .ip values in the system, perform one or more of the following:
   - Select a specific mismatched parameter, interface, or signal and click >> to synchronize the items.
   - Click Sync All to synchronize all values for the current component.
   - Click Sync All System Info to synchronize all IP components in the current system at once.

1.13. Generating a Platform Designer System

Platform Designer system generation creates the interconnect between IP components, and generates files for Intel Quartus Prime synthesis and simulation in supported third-party tools.

Follow these steps to generate a Platform Designer system:
1. Open a system in Platform Designer.
2. Consider whether to specify a unique generation ID, as Specifying the Generation ID on page 54 describes.
3. Click the Generate HDL button. The Generation dialog box appears.
4. Specify options for generation of **Synthesis**, **Simulation**, and testbench files, as **Generation Dialog Box Options** on page 54 describes.

5. Consider whether to specify options for **Parallel IP Generation**, as **Disabling or Enabling Parallel IP Generation** describes.

6. To start system generation, click **Generate**.
   
   Note: Platform Designer may add unique suffixes (hashes) to ip component files during generation to ensure uniqueness of the file. The uniqueness of the files is necessary because the IP component is dynamic. The RTL generates during runtime, according to the input parameters. This methodology ensures no collisions between the multiple variants of the same IP. The hash derives from the parameter values that you specify. A given set of parameter values produces the same hash for each generation.

### 1.13.1. Generation Dialog Box Options

Platform Designer system generation creates files for Intel Quartus Prime synthesis and supported third-party simulators. The **Generation** dialog box appears when you click **Generate HDL**, or when you attempt to close a system prior to generation.

You can specify the following system generation options in the **Generation** dialog box:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create HDL design files for synthesis</strong></td>
<td>Allows you to specify Verilog or VHDL file type generation for the system’s top-level definition and child instances. Select <strong>None</strong> to skip generation of synthesis files.</td>
</tr>
<tr>
<td><strong>Create timing and resource estimates for each IP in your system to be used with third-party synthesis tools</strong></td>
<td>Generates a non-functional Verilog Design File (.v) for use by supported third-party EDA synthesis tools. Estimates timing and resource usage for the IP component. The generated netlist file name is <code>&lt;ip_component_name&gt;_syn.v</code>.</td>
</tr>
<tr>
<td><strong>Create Block Symbol File (.bsf)</strong></td>
<td>Generates a Block Symbol File (.bsf) for use in a larger system schematic Block Diagram File (.bdf).</td>
</tr>
<tr>
<td><strong>Generate IP Core Documentation</strong></td>
<td>Generates the IP user guide documentation for the components in your system (when available).</td>
</tr>
</tbody>
</table>
| **Create simulation model**                        | Allows you to generate Verilog HDL or VHDL simulation model and simulation script files.  
   
   **Note:** ModelSim* - Intel FPGA Edition supports native, mixed-language (VHDL/Verilog/SystemVerilog) simulation. Therefore, Intel simulation libraries may not be compatible with single language simulators. If you have a VHDL-only license, some versions of ModelSim simulators may not support simulation for IPs written in Verilog. As a workaround, you can use ModelSim - Intel FPGA Edition, or purchase a mixed language simulation license from Mentor. |
| **Path**                                           | Specifies the output directory path.                                        |

### 1.13.2. Specifying the Generation ID

You can specify the **Generation ID** to uniquely identify that specific system generation. This parameter allows system tools, such as Nios II or HPS (Hard Processor System), to verify software-build compatibility with a specific Platform Designer system.
The **Generation ID** parameter is a unique integer value that derives from the timestamp during Platform Designer system generation. You can optionally modify this value to a value of your choosing to identify the system.

To specify the **Generation ID** parameter:
1. In the **Hierarchy** tab, select the top-level system.
2. Click **View ➤ Parameters**.
3. Under **System Identifier**, view or edit the value of **Generation ID**.

**Figure 28. Generation ID in Parameters Tab**

![Generation ID in Parameters Tab](image)

### 1.13.3. Files Generated for IP Cores and Platform Designer Systems

The Intel Quartus Prime Standard Edition software generates one of the following output file structures for individual IP cores that use one of the legacy parameter editors.
1.13.4. Generating System Testbench Files

Platform Designer can generate testbench files that instantiate the current Platform Designer system and add Bus Functional Models (BFMs) to drive the top-level interfaces. BFMs interact with the system in the simulator.
You can generate a standard or simple testbench system with BFM or Mentor Verification IP (for AMBA 3 AXI or AMBA 4 AXI) components that drive the external interfaces of the system. Platform Designer generates a Verilog HDL or VHDL simulation model for the testbench system to use in the simulation tool.

First generate a testbench system, and then modify the testbench system in Platform Designer before generating the simulation model. Typically, you select only one of the simulation model options.

Follow these steps to generate system testbench files:

1. Open and configure a system in Platform Designer.
2. Click **Generate** ➤ **Generate Testbench System**. The **Generation** dialog box appears.
3. Specify options for the test bench system:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| Create testbench Platform Designer system | Specifies a simple or standard testbench system:  
- **Standard, BFMs for standard Platform Designer Interconnect**—Creates a testbench Platform Designer system with BFM IP components attached to exported Avalon and AMBA 3 AXI or AMBA 3 AXI interfaces. Includes any simulation partner modules specified by IP components in the system. The testbench generator supports AXI interfaces and can connect AMBA 3 AXI or AMBA 3 AXI interfaces to Mentor Graphics AMBA 3 AXI or AMBA 3 AXI master/slave BFMs. However, BFMs support address widths only up to 32-bits.  
- **Simple, BFMs for clocks and resets**—Creates a testbench Platform Designer 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 | Specifies Verilog HDL or VHDL simulation model files and simulation scripts for the testbench. Use this option if you do not need to modify the Platform Designer-generated testbench before running the simulation. |
| Output directory               | Specifies the path for output of generated testbench files. Turn on **Clear output** to remove any previously generated content from the location.                                                                 |
| Parallel IP Generation         | Turn on **Use multiple processors for faster IP generation (when available)** to generate IP using multiple CPUs when available in your system.                                                                |

4. Click **Generate**. The testbench files generate according to your specifications.
5. Open the testbench system in Platform Designer. 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 Avalon Interrupt Source Intel FPGA IP component.
6. If you modify a BFM, regenerate the simulation model for the testbench system.
7. Compile the system and load the Platform Designer system and testbench into your simulator, and then run the simulation.

### 1.13.4.1. Platform Designer Testbench Simulation Output Directories

Platform Designer generates the following testbench files.
1.13.4.2. Platform Designer Testbench Files

Platform Designer generates the following testbench files.

Table 10. Platform Designer 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 Platform Designer testbench system.</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>
</tbody>
</table>

continued...
File Name or Directory Name | Description
--- | ---
<system>_tb.spd | Required input file for `ip-make-simscript` to generate simulation scripts for supported simulators. The .spd file contains a list of files generated for simulation and information about memory that you can initialize.

<system>.html and <system>_tb.html | A system report that contains connection information, a memory map showing the address of each slave with respect to each master to which it is connected, and parameter assignments.

<system>_generation.rpt | Platform Designer generation log file. A summary of the messages that Platform Designer issues during testbench system generation.

<system>.ipx | The IP Index File (.ipx) lists the available IP components, or a reference to other directories to search for IP components.

<system>.svd | Allows HPS System Debug tools to view the register maps of peripherals connected to HPS within a Platform Designer system. Similarly, during synthesis the .svd files for slave interfaces visible to System Console masters are stored in the .sof file in the debug section. System Console reads this section, which Platform Designer can query for register map information. For system slaves, Platform Designer can access the registers by name.

mentor/ | Contains a ModelSim script msim_setup.tcl to set up and run a simulation

aldec/ | Contains a Riviera-PRO* script rivierapro_setup.tcl to setup and run a simulation.

/synopsys/vcs /synopsys/vcsmx | Contains a shell script vcs_setup.sh to set up and run a VCS* simulation. Contains a shell script vcsmx_setup.sh and synopsys_sim.setup file to set up and run a VCS MX simulation.

/cadence | Contains a shell script ncsim_setup.sh and other setup files to set up and run an NCSIM simulation.

/submodules | Contains HDL files for the submodule of the Platform Designer testbench system.

<child IP cores>/ | For each generated child IP core directory, Platform Designer testbench generates /synth and /sim subdirectories.

### 1.13.5. Generating Example Designs for IP Components

Some Platform Designer IP components include example designs that you can use or modify to replicate similar functionality in your own system. You must generate the examples to view or use them.

Use any of the following methods to generate example designs for IP components:

- Double-click the IP component in the Platform Designer IP Catalog or **System Contents** tab. The parameter editor for the component appears. If available, click the **Example Design** button in the parameter editor to generate the example design. The **Example Design** button only appears in the parameter editor if an example is available.

- For some IP components, click **Generate ➤ Generate Example Design** to access an example design. This command only enables when a design example is available.

The following Platform Designer system example designs demonstrate various design features and flows that you can replicate in your Platform Designer system.

**Related Information**

Intel FPGA Design Example Web Page
1.13.6. Generating the HPS IP Component System View Description File

Platform Designer 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) that Arm provides. 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 Platform Designer system.

Related Information
- Component Interface Tcl Reference on page 463
- CMSIS - Cortex Microcontroller Software

1.13.7. Generating Header Files for Master Components

You can use the sopc-create-header-files command from the Nios II command shell to create header files for any master component in your Platform Designer system. The Nios II tool chain uses this command to create the processor's system.h file. You can also use this command to generate system level information for a hard processing system (HPS) in Intel's SoC devices or other external processors. The header file includes address map information for each slave, relative to each master that accesses the slave. Different masters may have different address maps to access a particular slave component. By default, the header files are in C format and have a .h suffix. You can select other formats with appropriate command-line options.

Table 11. sopc-create-header-files Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sopc&gt;</code></td>
<td>Path to Platform Designer .sopcinfo file, or the file directory. If you omit this option, the path defaults to the current directory. If you specify a directory path, you must make sure that there is a .sopcinfo file in the directory.</td>
</tr>
<tr>
<td>--separate-masters</td>
<td>Does not combine a module's masters that are in the same address space.</td>
</tr>
<tr>
<td>--output-dir[=dirname]</td>
<td>Allows you to specify multiple header files in dirname. The default output directory is '.'</td>
</tr>
<tr>
<td>--single[=filename]</td>
<td>Allows you to create a single header file, filename.</td>
</tr>
<tr>
<td>--single-prefix[=prefix]</td>
<td>Prefixes macros from a selected single master.</td>
</tr>
<tr>
<td>--module[=moduleName]</td>
<td>Specifies the module name when creating a single header file.</td>
</tr>
<tr>
<td>--master[=masterName]</td>
<td>Specifies the master name when creating a single header file.</td>
</tr>
<tr>
<td>--format[=type]</td>
<td>Specifies the header file format. Default file format is .h.</td>
</tr>
<tr>
<td>--silent</td>
<td>Does not display normal messages.</td>
</tr>
<tr>
<td>--help</td>
<td>Displays help for sopc-create-header-files.</td>
</tr>
</tbody>
</table>

By default, the sopc-create-header-files command creates multiple header files. There is one header file for the entire system, and one header file for each master group in each module. A master group is a set of masters in a module in the same address space. In general, a module may have multiple master groups. Addresses and available devices are a function of the master group.
Alternatively, you can use the `--single` option to create one header file for one master group. If there is one CPU module in the Platform Designer system with one master group, the command generates a header file for that CPU's master group. If there are no CPU modules, but there is one module with one master group, the command generates the header file for that module's master group.

You can use the `--module` and `--master` options to override these defaults. If your module has multiple master groups, use the `--master` option to specify the name of a master in the desired master group.

<table>
<thead>
<tr>
<th>Table 12. Supported Header File Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>m4</td>
</tr>
<tr>
<td>sh</td>
</tr>
<tr>
<td>mk</td>
</tr>
<tr>
<td>pm</td>
</tr>
</tbody>
</table>

*Note:* You can use the `sopc-create-header-files` command when you want to generate C macro files for DMAs that have access to memory that the Nios II does not have access to.

### 1.14. Simulating a Platform Designer System

You can simulate a Platform Designer system in a supported third-party simulator to verify and debug operation. Platform Designer generates the simulation models for your system, along with optional scripts to set up the simulation environment for specific, supported third-party simulators.

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>
<thead>
<tr>
<th>Table 13. Simulation Script Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>TOP_LEVEL_NAME</td>
</tr>
<tr>
<td>QSYS_SIMDIR</td>
</tr>
<tr>
<td>QUARTUS_INSTALL_DIR</td>
</tr>
</tbody>
</table>
Example 4. Top-Level Simulation HDL File for a Testbench System

The example below shows the pattern_generator_tb generated for a Platform Designer 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 Tristate Conduit BFM component is not supported in Synopsys VCS, NCSim, and Riviera-PRO in the Intel 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.

1.14.1. Adding 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 - Intel FPGA Edition does not support SystemVerilog assertions. If you want to use assertion monitors, you must use a supported third-party simulator. For more information, refer to Introduction to Intel FPGA IP Cores.

Figure 31. 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.

1.14.2. 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 Platform Designer testbench system with the following steps:

1. Click Generate ➤ Generate Testbench System.
2. In the Generation dialog box, select Simple, BFMs for clocks and resets.
3. For Create testbench simulation model, select Verilog or VHDL.
4. Click Generate.
5. Open the Nios II Software Build Tools for Eclipse.
6. Set up an application project and board support package (BSP) for the <system>.sopcinfo file.
7. To simulate, right-click the application project in Eclipse, and then click **Run as ➤ Nios II ModelSim**. This command prepares the ModelSim simulation environment, and compiles and loads the Nios II software simulation.

8. To run the simulation in ModelSim, type `run -all` in the ModelSim transcript window.

9. Set the ModelSim settings and select the Platform Designer Testbench Simulation Package Descriptor (`.spd`) file, `<system>_tb.spd`. The `.spd` file generates with the testbench simulation model for Nios II designs, and specifies the files you require for Nios II simulation.

**Related Information**

Nios II Gen2 Software Developer's Handbook

### 1.15. Integrating a Platform Designer System with the Intel Quartus Prime Software

To integrate a Platform Designer system with your Intel Quartus Prime project, you must add either the Platform Designer System File (`.qsys`) or the Intel Quartus Prime IP File (`.qip`), but never both to your Intel Quartus Prime project. Platform Designer creates the `.qsys` file when you save your Platform Designer system, and produces the `.qip` file when you generate your Platform Designer system. Both the `.qsys` and `.qip` files contain the information necessary for compiling your Platform Designer system within an Intel Quartus Prime project.

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

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

**Note:**

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

### Does Intel Quartus Prime Overwrite Platform Designer-Generated Files During Compilation?

Platform Designer supports standard and legacy device generation. Standard device generation refers to generating files for the Intel Arria® 10 device, and later device families. Legacy device generation refers to generating files for device families prior to the release of the Intel Arria 10 device, including MAX 10 devices.

When you integrate your Platform Designer system with the Intel Quartus Prime software, if a `.qsys` file is included as a source file, Platform Designer 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, Platform Designer generates HDL files in the Intel Quartus Prime project directory under `/db/ip`. 
For standard devices, Platform Designer-generated files are only overwritten during Intel Quartus Prime compilation if the .qip file is removed or missing. For legacy devices, each time you compile your Intel Quartus Prime project with a .qsys file, the Platform Designer-generated files are overwritten. Therefore, you should not edit Platform Designer-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

- Generating a Platform Designer System on page 53
- IP Core Generation Output
- Introduction to Intel FPGA IP Cores
- Implementing and Parameterizing Memory IP

1.15.1. Integrate a Platform Designer System and the Intel Quartus Prime Software With the .qsys File

Use the following steps to integrate your Platform Designer system and your Intel Quartus Prime project using the .qsys file:

1. In Platform Designer, create and save a Platform Designer system.

2. To automatically include the .qsys file in the your Intel Quartus Prime project during compilation, in the Intel Quartus Prime software, select Tools ➤ Options ➤ IP Settings, and turn on Automatically add Intel Quartus Prime IP files to all projects.

3. When the Automatically add Intel Quartus Prime IP files to all projects option is not checked, when you exit Platform Designer, the Intel Quartus Prime software displays a dialog box asking whether you want to add the .qsys file to your Intel Quartus Prime project. Click Yes to add the .qsys file to your Intel Quartus Prime project.

4. In the Intel Quartus Prime software, select Processing ➤ Start Compilation.

1.15.2. Integrate a Platform Designer System and the Intel Quartus Prime Software With the .qip File

Use the following steps to integrate your Platform Designer system and your Intel Quartus Prime project using the .qip file:

1. In Platform Designer, create and save a Platform Designer system.

2. In Platform Designer, click Generate HDL.

3. In the Intel 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 Intel Quartus Prime project.

5. In the Intel Quartus Prime software, select Processing ➤ Start Compilation.
1.16. Managing Hierarchical Platform Designer Systems

Platform Designer supports hierarchical systems that include one or more Platform Designer subsystems within another Platform Designer system. Platform Designer allows you to create, explore, and edit systems and subsystems together in the same Platform Designer window. Platform Designer generates the complete system hierarchy during the top-level system's generation.

All hierarchical Platform Designer systems appear in the IP Catalog under Project ➤ System. You select the system from the IP Catalog to reuse the system across multiple designs. In a team-based hierarchical design flow, you can divide large designs into subsystems and allow team members develop subsystems simultaneously.

Related Information
Viewing the System Hierarchy on page 13

1.16.1. Adding a Subsystem to a Platform Designer System

You can add a Platform Designer system as a subsystem (child) of another Platform Designer system (parent), at any level in the parent system hierarchy.

Follow these steps to add a subsystem to a Platform Designer system:

1. Create a Platform Designer system to use as the subsystem.
2. Open a Platform Designer system to contain the subsystem.
3. On the System Contents tab, use any of the following methods to add the subsystem:
   • Right-click anywhere in the System Contents and click Add a new subsystem to the current system.
   • Click the Add a new subsystem to the current system button on the toolbar.
   • Press Ctrl+Shift+N.
4. In the Confirm New System Name dialog box, confirm or specify the new system file name and click OK. The system appears as a new subsystem in the System Contents.
1.16.2. Viewing and Traversing Subsystem Contents

You can view and traverse the elements and connections within subsystems in a hierarchical Platform Designer system.

Follow these steps to view and traverse subsystem contents:
1. Open a Platform Designer system that contains a subsystem.
2. Use any of the following methods to view the subsystem contents:
   - Double-click a subsystem in the Hierarchy tab. The subsystem opens in the System Contents.
   - Right-click a system in the Hierarchy, System Contents, or Schematic tabs, and then select Drill into subsystem.
   - Press Ctrl+Shift+D in the System Contents tab.
3. Use any of the following System Contents or Schematic tab toolbar buttons to traverse the system and subsystems:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![button]</td>
<td>Move to the top of the hierarchy—navigates to the top-level (parent) .qsys file for the system.</td>
</tr>
<tr>
<td>![button]</td>
<td>Move up one level of hierarchy—navigates up one hierarchy level from the current selection.</td>
</tr>
<tr>
<td>![button]</td>
<td>Drill into a subsystem to explore its contents—opens the subsystem you select in the System Contents.</td>
</tr>
</tbody>
</table>

**Note:** In the System Contents tab, you can press Ctrl+Shift+U to navigate up one level, and Ctrl+Shift+D to drill into a system.
1.16.3. Editing a Subsystem

You can double-click a Platform Designer 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 the system into another subsystem with commands in the System Contents tab.

Note: You can only edit subsystems that a writable .qsys file preserves. You cannot edit systems that you create from composed _hw.tcl files, or systems that define instance parameters.

Follow these steps to edit a Platform Designer subsystem:

1. Open a Platform Designer system that contains a subsystem.
2. In the System Contents or Schematic tabs, use the Move Up, Move Down, Move to Top, and Move to Bottom toolbar buttons to navigate the system level you want to edit. Platform Designer updates to reflect your selection.
3. To edit a system, double-click the system in the Hierarchy tab. The system opens and is available for edit in all Platform Designer views.
4. In the System Contents tab, you can rename any element, add, remove, or duplicate connections, and export interfaces, as appropriate.

Note: Changes to a subsystem affect all instances. Platform Designer identifies unsaved changes to a subsystem with an asterisk next to the subsystem in the Hierarchy tab.

1.16.4. Changing a Component's Hierarchy Level

You can change the hierarchical level of components in your system.
You can lower the hierarchical level of a component, even into its own subsystem, which can simplify the top-level system view. You can also raise the level of a component or subsystem to share the component or subsystem between two unique subsystems. Management of hierarchy levels facilitates system optimization and can reduce complex connectivity in your subsystems.

Follow these steps to change a component’s hierarchy level:

1. Open a Platform Designer system that contains a subsystem.
2. In the System Contents tab, to group and change the hierarchy level of multiple components that share a system-level component, multi-select the components, right-click, and then click **Push down into new subsystem**. Platform Designer pushes the components into their own subsystem and re-establishes the exported signals and connectivity in the new location.
3. In the System Contents tab, to pull a component up out of a subsystem, select the component, and then click **Pull up**. Platform Designer pulls the component up out of the subsystem and re-establishes the exported signals and connectivity in the new location.

### 1.16.5. Saving a Subsystem

When you save a subsystem as part of a Platform Designer system, Platform Designer confirms the new subsystem name in the **Confirm New System Filenames** dialog box. By default, Platform Designer suggests the same name as the subsystem .qsys file and saves in the project’s /ip directory.

Follow these steps to save a subsystem:

1. Open a Platform Designer system that contains a subsystem.
2. Click File ➤ Save to save your Platform Designer design.
3. 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 new name, and then press Enter, to move to the next un-named system.
4. In the **Confirm New System Filenames** dialog box, to edit the name of a subsystem, click the subsystem, and then type the new name.

### 1.16.6. Exporting a System as an IP Component

You can export a Platform Designer system as a _hw.tcl component for use in other Platform Designer systems.

1. Open a Platform Designer system.
2. Click File ➤ Export System as hw.tcl Component.

The exported system displays as a new component under the **System** category in the IP Catalog.
1.16.7. 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 Platform Designer 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 Platform Designer system. With your completed system, you can vary the values of the instance parameters to review their effect within the On-Chip Memory component.

1.16.7.1. Create the Memory System

This procedure creates a Platform Designer system to use as subsystem as part of a hierarchical instance parameter example.

1. In Platform Designer, 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,** Platform Designer 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 Platform Designer system as memory_system.qsys.
1.16.7.2. Add Platform Designer 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 Platform Designer 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 Platform Designer 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.

**Figure 37. Preview an Instance in the Parameter Editor**

10. Click **File ➤ Save**.
1.16.7.3. Create a Platform Designer Instantiating Memory System

This procedure creates a Platform Designer system to use as a higher-level system as part of a hierarchical instance parameter example.

1. In Platform Designer, 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, Platform Designer 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 Platform Designer as instantiating_memory_system.qsys.

Figure 38. Instantiating Memory System (instantiating_memory_system.qsys)

1.16.7.4. Apply Instance Parameters at a Higher-Level Platform Designer System and Pass the Parameters to the Instantiated Lower-Level System

This procedure shows you how to use Platform Designer 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.
1.17. Creating a System with Platform Designer Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019.05.14</td>
<td>18.1.0</td>
<td>• Removed &quot;Wire-Level Connectivity&quot; section. This description was misplaced and applies only to Intel Quartus Prime Pro Edition.</td>
</tr>
</tbody>
</table>
| 2018.12.15       | 18.1.0                     | • Moved command-line utility information into new "Platform Designer Command-Line Interface" chapter.
                          • Revised headings and re-organized content into user task-based sections. |
                          • Removed duplicated topic: Manually Control Pipelining in the Platform Design Interconnect. The topic is now in the Platform Design Interconnect chapter.
                          • Reorganized information about associating Intel Quartus Prime projects to Platform Designer systems.
                          • Grouped information regarding definition and management of IP cores in Platform Designer under topic: IP Cores in Platform Designer, and updated contents.
                          • In topic 64-Bit Addressing Support, added link to information about the auto base assignment feature. |
| 2017.11.06       | 17.1.0                     | • Changed instances of Qsys to Platform Designer (Standard) |
| 2016.05.03       | 16.0.0                     | • Qsys Command-Line Utilities updated with latest supported command-line options.
                          • Added: Generate Header Files |
| 2015.11.02       | 15.1.0                     | • Added: Troubleshooting IP or Qsys System Upgrade.
                          • Added: Generating Version-Agnostic IP and Qsys Simulation Scripts.
                          • Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04       | 15.0.0                     | • New figure: Avalon-MM Write Master Timing Waveforms in the Parameters Tab.
                          • Added **Enable ECC protection** option, Specify Qsys Interconnect Requirements.
                          • Added External Memory Interface Debug Toolkit note, Generate a Qsys System.
                          • Modelsim- Altera now supports native mixed-language (VHDL/Verilog/ SystemVerilog) simulation, Generating Files for Synthesis and Simulation. |
| December 2014    | 14.1.0                     | • Create and Manage Hierarchical Qsys Systems.
                          • Schematic tab.
                          • View and Filter Clock and Reset Domains.
                          • File ➤ Recent Projects menu item.
                          • Updated example: Hierarchical System Using Instance Parameters |
| August 2014      | 14.0a10.0                  | • Added distinction between legacy and standard device generation.
                          • Updated: Upgrading Outdated IP Components.
                          • Updated: Generating a Qsys System.
                          • Updated: Integrating a Qsys System with the Quartus II Software.
                          • Added screen shot: Displaying Your Qsys System. |
| June 2014        | 14.0.0                     | • Added tab descriptions: Details, Connections.
                          • Added Managing IP Settings in the Quartus II Software.
                          • Added Upgrading Outdated IP Components.
                          • Added Support for Avalon-MM Non-Power of Two Data Widths. |

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| November 2013    | 13.1.0                      | • Added Integrating with the .qsys File.  
|                  |                             | • Added Using the Hierarchy Tab.         
|                  |                             | • Added Managing Interconnect Requirements.  
|                  |                             | • Added Viewing Qsys Interconnect.        |
| May 2013         | 13.0.0                      | • Added AMBA APB support.                
|                  |                             | • Added qsys-generate utility.           
|                  |                             | • Added VHDL BFM ID support.             
|                  |                             | • Added Creating Secure Systems (TrustZones) .  
|                  |                             | • Added CMSIS Support for Qsys Systems With An HPS Component.  
|                  |                             | • Added VHDL language support options.          |
| November 2012    | 12.1.0                      | • Added AMBA AXI4 support.               |
| June 2012        | 12.0.0                      | • Added AMBA AXI3 support.               
|                  |                             | • Added Preset Editor updates.           
|                  |                             | • Added command-line utilities, and scripts. |
| November 2011    | 11.1.0                      | • Added Synopsys VCS and VCS MX Simulation Shell Script.  
|                  |                             | • Added Cadence Incisive Enterprise (NCSIM) Simulation Shell Script.  
|                  |                             | • Added Using Instance Parameters and Example Hierarchical System Using Parameters. |
| May 2011         | 11.0.0                      | • Added simulation support in Verilog HDL and VHDL.  
|                  |                             | • Added testbench generation support.     
|                  |                             | • Updated simulation and file generation sections. |
| December 2010    | 10.1.0                      | Initial release.                        |

**Related Information**

**Documentation Archive**

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.
2. Optimizing Platform Designer System Performance

Platform Designer provides tools that allow you to optimize the performance of the system interconnect for Intel FPGA designs. This chapter presents techniques that leverage the available tools and the trade-offs of each implementation.

Note: Intel now refers to Qsys as Platform Designer (Standard).

The foundation of any system is the interconnect logic that connects hardware blocks or components. Creating interconnect logic is time consuming and prone to errors, and existing interconnect logic is difficult to modify when design requirements change. The Platform Designer system integration tool addresses these issues and provides an automatically generated and optimized interconnect designed to satisfy the system requirements.

Platform Designer supports Avalon, AMBA 3 AXI (version 1.0), AMBA 4 AXI (version 2.0), AMBA 4 AXI-Lite (version 2.0), AMBA 4 AXI-Stream (version 1.0), and AMBA 3 APB (version 1.0) interface specifications.

Note: Recommended Intel practices may improve clock frequency, throughput, logic utilization, or power consumption of a Platform Designer design. When you design a Platform Designer system, use your knowledge of the design intent and goals to further optimize system performance beyond the automated optimization available in Platform Designer.

Related Information
• Creating a System with Platform Designer on page 10
• Creating Platform Designer Components on page 282
• Platform Designer Interconnect on page 124
• Avalon Interface Specifications
• AMBA Protocol Specifications

2.1. Designing with Avalon and AXI Interfaces

Platform Designer 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 pair of components is connected without any requirement to arbitrate between the data source and sink.

Because Platform Designer 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 Platform Designer Components on page 282

2.1.1. 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 deasserted, 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.

2.1.2. 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 four 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.

### 2.2. 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 Platform Designer 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 Avalon-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.

**Figure 42. Avalon-MM Pipeline Bridge**
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

### 2.3. Using Concurrency in Memory-Mapped Systems

Platform Designer interconnect uses parallel hardware in FPGAs, which allows you to design concurrency into your system and process transactions simultaneously.
2.3.1. Implementing Concurrency With Multiple Masters

Implementing concurrency requires multiple masters in a Platform Designer 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 Platform Designer generates an interconnect with slave-side arbitration, every master interface in a system can issue transfers concurrently, if they are not posting transfers to the same slave. Concurrency is limited by the number of master interfaces sharing any particular slave interface. If a design requires higher data throughput, you can increase the number of master and slave interfaces to increase the number of transfers that occur simultaneously. The example below shows a system with three master interfaces.

**Figure 45.** Avalon Multiple Master Parallel Access

In this Avalon example, the DMA engine operates with Avalon-MM read and write masters. The yellow lines represent active simultaneous connections.
**Figure 46. AXI Multiple Master Parallel Access**

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

2.3.2. 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.
2.3.3. 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 48. Single or Dual DMA Channels

Single DMA Channel
Maximum of One Read & One Write Per Clock Cycle

Dual DMA Channels
Maximum of Two Reads & Two Writes Per Clock Cycle
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.

### 2.4. Inserting Pipeline Stages to Increase System Frequency

Adding pipeline stages may increase the $f_{\text{MAX}}$ of the design by reducing the combinational logic depth, at the cost of additional latency and logic utilization.

Platform Designer provides the Limit interconnect pipeline stages to option on the Interconnect Requirements tab to automatically add pipeline stages to the Platform Designer interconnect when you generate a system.

The Limit interconnect pipeline stages to parameter in the Interconnect Requirements tab allows you to define the maximum Avalon-ST pipeline stages that Platform Designer can insert during generation. You can specify between 0 to 4 pipeline stages, where 0 means that the interconnect has a combinational datapath. You can specify a unique interconnect pipeline stage value for each subsystem.

For more information, refer to Interconnect Pipelining.

**Related Information**

Pipelined Avalon-MM Interfaces on page 102

### 2.5. Using Bridges

You can use bridges to increase system frequency, minimize generated Platform Designer logic, minimize adapter logic, and to structure system topology when you want to control where Platform Designer 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 Platform Designer 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. Bridges can provide more control over interconnect pipelining than the Limit interconnect pipeline stages to option.
Note: You can use Avalon bridges between AXI interfaces, and between Avalon domains. Platform Designer 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 Platform Designer Interconnect.

Related Information
- Bridges on page 210
- AMBA 3 APB Protocol Specification Support (version 1.0) on page 188

2.5.1. Using Bridges to Increase System Frequency

In Platform Designer, 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.

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

The Avalon-MM pipeline bridge component integrates into any Platform Designer 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 49. Avalon-MM Pipeline Bridge
2.5.1.1.1. Implementing Command Pipelining (Master-to-Slave)

When multiple masters share a slave device, you can use command pipelining to improve performance.

The arbitration logic for the slave interface must multiplex the address, writedata, and burstcount signals. The multiplexer width increases proportionally with the number of masters connecting to a single slave interface. The increased multiplexer width may become a timing critical path in the system. If a single pipeline bridge does not provide enough pipelining, you can instantiate multiple instances of the bridge in a tree structure to increase the pipelining and further reduce the width of the multiplexer at the slave interface.
2.5.1.1.2. Implementing Response Pipelining (Slave-to-Master)

When masters connect to multiple slaves that support read transfers, you can use slave-to-master pipelining to improve performance.
The interconnect inserts a multiplexer for every read datapath 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 $f_{\text{MAX}}$.

2.5.1.2. 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 Intel 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 datapaths.

2.5.2. Using Bridges to Minimize Design Logic

Bridges can reduce interconnect logic by reducing the amount of arbitration and multiplexer logic that Platform Designer generates. This reduction occurs because bridges limit the number of concurrent transfers that can occur.

2.5.2.1. Avoiding Speed Optimizations That Increase Logic

You can add an additional pipeline stage with a pipeline bridge between masters and slaves to reduce 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 Intel 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.
2.5.2.2. Limiting Concurrency

The amount of logic generated for the interconnect often increases as the system becomes larger because Platform Designer creates arbitration logic for every slave interface that is shared by multiple master interfaces. Platform Designer inserts multiplexer logic between master interfaces that connect to multiple slave interfaces if both support read datapaths.

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 datapath 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, Platform Designer generates three arbiters and three multiplexers for the read datapath. 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. Platform Designer creates one arbitration block between the bridge and the three masters, and a single read datapath 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 datapaths to ensure that you do not limit overall system performance.
2.5.3. Using Bridges to Minimize Adapter Logic

Platform Designer 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.

Platform Designer 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 Platform Designer generates.

2.5.3.1. 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 \(2^{(\text{width}(\text{burstcount} - 1))}\), therefore, if the \text{burstcount} width is four bits, the maximum burst length is eight. If no \text{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 \text{Clock} column for the master and slave interfaces. If the clock is different for the master and slave interfaces, Platform Designer 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 Platform Designer 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.

### 2.5.3.2. Changing the Response Buffer Depth

When you use automatic clock-crossing adapters, Platform Designer determines the required depth of FIFO buffering based on the slave properties. If a slave has a high \text{Maximum Pending Reads} parameter, the resulting deep response buffer FIFO that Platform Designer 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 \text{Maximum Pending Reads} available from the slave.

### 2.5.4. 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 determine the impact to the design.

#### 2.5.4.1. 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 not be acceptable in your design.

**2.5.4.1.1. Acceptable Latency Increase**

For a pipeline bridge, Platform Designer 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 52. 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 53. High Efficiency Read Transfer**

### 2.5.4.1.2. 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 datapath 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.
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.

### 2.5.4.2. 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, Platform Designer 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.
Inappropriate Use of a Bridge in a Hierarchical System

A memory subsystem with one bridge that acts as a single slave interface for the Avalon-MM Nios II and DMA masters, which results in a bottleneck architecture. The bridge acts as a bottleneck between the two masters and the memories.

If the $f_{\text{MAX}}$ 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.
2.5.4.3. Address Space Translation

The slave interface of a pipeline or clock crossing bridge has a base address and address span. You can set the base address, or allow Platform Designer 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.

### 2.5.4.4. 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, Platform Designer 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 60. Address Translation Corrected With Bridge**

2.6. 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 Platform Designer 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
2.6.1. 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.

2.6.1.1. 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. Platform Designer 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 if 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.

### 2.6.2. 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
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 access to another master. 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**

*Arbitration* on page 136

### 2.6.2.1. Differences Between Arbitration Shares and Bursts

The following three key characteristics distinguish arbitration shares and bursts:

- Arbitration Lock
- Sequential Addressing
- Burst Adapters

**Arbitration Lock**

When a master posts a burst transfer, the arbitration is locked for that master;
consequently, the bursting master should be capable of sustaining transfers for the
duration of the locked period. If, after the fourth write, the master deasserts the write
signal (Avalon-MM write or AXI `wvalid`) for fifty cycles, all other masters continue to
wait for access during this stalled period.

To avoid wasted bandwidth, your master designs should wait until a full burst transfer
is ready before requesting access to a slave device. Alternatively, you can avoid
wasted bandwidth by posting `burstcounts` equal to the amount of data that is ready.
For example, if you create a custom bursting write master with a maximum burstcount of eight, but only three words of data are ready, you can present a burstcount of three. This strategy does not result in optimal use of the system 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 burstcount, which represents the number of words of data that are transferred, starting from the base address and incrementing sequentially. Burst transfers are common for processors, DMAs, and buffer processing accelerators; however, sometimes a master must access non-sequential addresses. Consequently, a bursting master must set the burstcount to the number of sequential addresses, and then reset the burstcount for the next location.

The arbitration share algorithm has no restrictions on addresses; therefore, your custom master can update the address it presents to the interconnect for every read or write transaction.

**Burst Adapters**

Platform Designer 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 Platform Designer generating burst adapters when appropriate.

Platform Designer 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. Platform Designer assigns non-bursting masters and slave interfaces a burst length of one. The burst adapter divides long bursts into shorter bursts. As a result, the burst adapter adds logic to the address and burstcount paths between the master and slave interfaces.

### 2.6.2.2. 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.

#### 2.6.2.2.1. 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 Platform Designer, the PIO, UART, and Timer include slave interfaces that use simple transfers.

#### 2.6.2.2.2. 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, Platform Designer 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, the interconnect contains logic to reconcile the differences.

### Table 15. 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>Platform Designer 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>Platform Designer 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>Platform Designer 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>Platform Designer 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>

#### 2.6.2.2.3. 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.
2.6.2.3. Avalon-MM Burst Master Example

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 \texttt{go} is asserted, the \texttt{start_address} and \texttt{transfer_length} are registered. On the next clock cycle, the control logic asserts \texttt{burst_begin}, which synchronizes the internal control signals in addition to the \texttt{master_address} and \texttt{master_burstcount} presented to the interconnect. The timing of these two signals is important because during bursting write transfers \texttt{byteenable} and \texttt{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 \texttt{waitrequest}. In this example, the FIFO's \texttt{used} signal tracks the number of words of data that are stored in the FIFO and determines when enough data has been buffered.

The \texttt{address} register increments after every word transfer, and the \texttt{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.
2.7. Reducing Logic Utilization

You can minimize logic size of Platform Designer systems. Typically, there is a trade-off between logic utilization and performance. Reducing logic utilization applies to both Avalon and AXI interfaces.

2.7.1. Minimizing Interconnect Logic to Reduce Logic Unitization

In Platform Designer, changes to the connections between master and slave reduce the amount of interconnect logic required in the system.

2.7.1.1. 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.

2.7.1.2. Removing Unnecessary Connections to Minimize Interconnect Logic

The number of connections between master and slave interfaces affects the $f_{MAX}$ of your system. Every master interface that you connect to a slave interface increases the width of the multiplexer width. As a multiplexer width increases, so does the logic depth and width that implements the multiplexer in the FPGA. To improve system performance, connect masters and slaves only when necessary.

When you connect a master interface to many slave interfaces, the multiplexer for the read data signal grows. Avalon typically uses a `readdata` signal. AXI read data signals add a response status and last indicator to the read response channel using `rdata`, `rresp`, and `rlast`. Additionally, bridges help control the depth of multiplexers.

2.7.1.3. 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.
2.7.2. 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.

2.7.2.1. Logic Consolidation Trade-Offs

You should consider the following trade-offs before making modifications to your system or interfaces:

- Consider the impact on concurrency that results when you consolidate components. When a system has four master components and four slave interfaces, it can initiate four concurrent accesses. If you consolidate the four slave interfaces into a single interface, then the four masters must compete for access. Consequently, you should only combine low priority interfaces such as low speed parallel I/O devices if the combination does not impact the performance.

- Determine whether consolidation introduces new decode and multiplexing logic for the slave interface that the interconnect previously included. If an interface contains multiple read and write address locations, the interface already contains the necessary decode and multiplexing logic. When you consolidate interfaces, you typically reuse the decoder and multiplexer blocks already present in one of the original interfaces; however, combining interfaces may simply move the decode and multiplexer logic, rather than eliminate duplication.

- Consider whether consolidating interfaces makes the design complicated. If so, you should not consolidate interfaces.

Related Information

Using Concurrency in Memory-Mapped Systems on page 81

2.7.2.2. 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.
Platform Designer 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, Platform Designer 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, Platform Designer 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.
Mixed Bursting System with Bridges

To reduce the number of adapters, you can add pipeline bridges. The pipeline bridge, between the Nios II/f core and the peripherals that do not support bursts, eliminates three burst adapters from the previous example. A second pipeline bridge between the Nios II/f core and the DDR SDRAM, with its maximum burst size set to eight, eliminates another burst adapter, as shown below.

2.7.3. Reducing Logic Utilization With Multiple Clock Domains

You specify clock domains in Platform Designer on the System Contents tab. Clock sources can be driven by external input signals to Platform Designer, or by PLLs inside Platform Designer. Clock domains are differentiated based on the name of the clock. You can create multiple asynchronous clocks with the same frequency.

Platform Designer generates Clock Domain Crossing (CDC) logic 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. Platform Designer 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 64. Clock Crossing Adapter**

This example illustrates a clock domain adapter between one master and one slave. The synchronizer blocks use multiple stages of flipflops 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 Platform Designer 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.

Platform Designer 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. Platform Designer 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

**2.7.4. 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.

2.8. Reducing Power Consumption

Platform Designer provides various low power design changes that enable you to reduce the power consumption of the interconnect and custom components.

2.8.1. Reducing Power Consumption With Multiple Clock Domains

When you use multiple clock domains, you should put non-critical logic in the slower clock domain. Platform Designer automatically reconciles data crossing over asynchronous clock domains by inserting clock crossing logic (handshake or FIFO).

You can use clock crossing in Platform Designer 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 Platform Designer)
- Serial peripheral interface (SPI)
- EPCS controller
- Tristate bridge and the components connected to the bridge

By reducing the clock frequency of the components connected to the bridge, you reduce the dynamic power consumption of the design. Dynamic power is a function of toggle rates and decreasing the clock frequency decreases the toggle rate.
Figure 65. Reducing Power Utilization Using a Bridge to Separate Clock Domains
Platform Designer 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 Platform Designer 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 Platform Designer:

- **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**—Platform Designer 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
2.8.2. Reducing Power Consumption by Minimizing Toggle Rates

A Platform Designer 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

Platform Designer 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, Platform Designer 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.
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.

2.8.3. Reducing Power Consumption by Disabling Logic

There are typically two types of low power modes: volatile and non-volatile. A volatile low power mode holds the component in a reset state. When the logic is reactivated, the previous operational state is lost. A non-volatile low power mode restores the previous operational state. You can use either software-controlled or hardware-controlled sleep modes to disable a component in order to reduce power consumption.
Software-Controlled Sleep Mode

To design a component that supports software-controlled sleep mode, create a single memory-mapped location that enables and disables logic by writing a zero or one. You can use the register’s output as a clock enable or reset, depending on whether the component has non-volatile requirements. The slave interface must remain active during sleep mode so that the enable bit is set when the component needs to be activated.

If multiple masters can access a component that supports sleep mode, you can use the mutex core to provide mutually exclusive accesses to your component. You can also build in the logic to re-enable the component on the very first access by any master in your system. If the component requires multiple clock cycles to re-activate, then it must assert a wait request to prolong the transfer as it exits sleep mode.

Hardware-Controlled Sleep Mode

Alternatively, you can implement a timer in your component that automatically causes the component to enter a sleep mode based on a timeout value specified in clock cycles between read or write accesses. Each access resets the timer to the timeout value. Each cycle with no accesses decrements the timeout value by one. If the counter reaches zero, the hardware enters sleep mode until the next access.

Figure 67. Hardware-Controlled Sleep Components

This example provides a schematic for the hardware-controlled sleep mode. If restoring the component to an active state takes a long time, use a long timeout value so that the component is not continuously entering and exiting sleep mode. The slave interface must remain functional while the rest of the component is in sleep mode. When the component exits sleep mode, the component must assert the waitrequest signal until it is ready for read or write accesses.

Related Information

Mutex Core

2.9. Reset Polarity and Synchronization in Platform Designer

When you add a component interface with a reset signal, Platform Designer 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, Platform Designer interconnect automatically inverts polarities as needed.
Each Platform Designer 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 Platform Designer system with the **Synchronous edges** parameter. In the clock source or reset bridge component, set the value of the **Synchronous edges** parameter to one of the following, depending on how the reset is externally synchronized:
• **None**—There is no synchronization on this reset.
• **Both**—The reset is synchronously asserted and deasserted with respect to the input clock.
• **Deassert**—The reset is synchronously asserted with respect to the input clock, and asynchronously deasserted.

**Figure 70. Synchronous Edges Parameter**

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

**Figure 71. 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_i</td>
<td>Clock Input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clk_n</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>reset</td>
<td>Reset 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, Platform Designer inserts adapters to synchronize or invert resets if there are mismatches in polarity or synchronization between the source and destination. You can view inserted adapters on the Memory-Mapped Interconnect tab with the System ➤ Show System with Platform Designer Interconnect command.

**Figure 72. Platform Designer Interconnect**

2.10. Optimizing Platform Designer System Performance Design Examples

Avalon Pipelined Read Master Example on page 119

Multiplexer Examples on page 121

2.10.1. Avalon Pipelined Read Master Example

For a high throughput system using the Avalon-MM standard, you can design a pipelined read master that allows a system to issue multiple read requests before data returns. Pipelined read masters hide the latency of read operations by posting reads as frequently as every clock cycle. You can use this type of master when the address logic is not dependent on the data returning.

2.10.1.1. Avalon Pipelined Read Master Example Design Requirements

You must carefully design the logic for the control and datapaths 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 while waitrequest is deasserted. While read is asserted, the address presented to the interconnect is stored.

The datapath 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.

### 2.10.1.2. Expected Throughput Improvement

The throughput improvement that you can achieve with a pipelined read master is typically directly proportional to the pipeline depth of the interconnect and the slave interface. For example, if the total latency is two cycles, you can double the throughput by inserting a pipelined read master, assuming the slave interface also supports pipeline transfers. If either the master or slave does not support pipelined read transfers, then the interconnect asserts waitrequest until the transfer completes. You can also gain throughput when there are some cycles of overhead before a read response.

Where reads are not pipelined, the throughput is reduced. When both the master and slave interfaces support pipelined read transfers, data flows in a continuous stream after the initial latency. You can use a pipelined read master that stores data in a FIFO to implement a custom DMA, hardware accelerator, or off-chip communication interface.

**Figure 73. Pipelined Read Master**
This example shows a pipelined read master that stores data in a FIFO. The master performs word accesses that are word-aligned and reads from sequential memory addresses. The transfer length is a multiple of the word size.

When the go bit is asserted, the master registers the start_address and transfer_length signals. The master begins issuing reads continuously on the next clock cycle until the length register reaches zero. In this example, the word size is four bytes so that the address always increments by four, and the length decrements by four. The read signal remains asserted unless the FIFO fills to a predetermined level. The address register increments and the length register decrements if the length has not reached 0 and a read is posted.

The master posts a read transfer every time the read signal is asserted and the waitrequest is deasserted. The master issues reads until the entire buffer has been read or waitrequest is asserted. An optional tracking block monitors the done bit. When the length register reaches zero, some reads are outstanding. The tracking logic prevents assertion of done until the last read completes, and monitors the number of reads posted to the interconnect so that it does not exceed the space remaining in the readdata FIFO. This example includes a counter that verifies that the following conditions are met:

1. If a read is posted and readdatavalid is deasserted, the counter increments.
2. 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.

### 2.10.2. Multiplexer Examples

You can combine adapters with streaming components to create datapaths 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 datapath that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. The on-chip FIFO memory has an input clock frequency of 100 MHz, and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time, and the second 72.7 percent of the time. You must 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.
The diagram below illustrates a datapath 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 100 MHz 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 datapath 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.
2. Optimizing Platform Designer System Performance

2.11. Optimizing Platform Designer System Performance Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017.11.06</td>
<td>17.1.0</td>
<td>• Changed instances of Qsys to Platform Designer (Standard)</td>
</tr>
</tbody>
</table>
| 2015.11.02        | 15.1.0                     | • Added: Reset Polarity and Synchronization in Qsys.  
|                   |                            | • Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04        | 15.0.0                     | Multiplexer Examples, rearranged description text for the figures. |
| May 2013          | 13.0.0                     | AMBA APB support. |
| November 2012     | 12.1.0                     | AMBA AXI4 support. |
| June 2012         | 12.0.0                     | AMBA AXI3 support. |
| November 2011     | 11.1.0                     | New document release. |

Related Information

Documentation Archive

For previous versions of the Intel Quartus Prime Handbook, search the documentation archives.
3. Platform Designer Interconnect

Platform Designer interconnect is a high-bandwidth structure that allows you to connect IP components to other IP components with various interfaces.

Note: Intel now refers to Qsys as Platform Designer (Standard).

Platform Designer supports Avalon, AMBA 3 AXI (version 1.0), AMBA 4 AXI (version 2.0), AMBA 4 AXI-Lite (version 2.0), AMBA 4 AXI-Stream (version 1.0), and AMBA 3 APB (version 1.0) interface specifications.

The video AMBA AXI and Intel Avalon Interoperation Using Platform Designer describes seamless integration of IP components using the AMBA AXI and the Intel Avalon interfaces.

Note: In Platform Designer systems with no clock domain crossing, the initial reset requires asserting for at least 16 cycles. This action prevents the propagation of incorrect values that the reset tree skew may generate during the initial reset release, ensuring the resetting of all the Platform Designer components and interconnect.

Related Information

- Avalon Interface Specifications
- Creating a System with Platform Designer on page 10
- Creating Platform Designer Components on page 282
- Platform Designer System Design Components on page 210
- AMBA AXI and Intel Avalon Interoperation Using Platform Designer
- Specifying Interconnect Requirements on page 35

3.1. Memory-Mapped Interfaces

Platform Designer supports the implementation of memory-mapped interfaces for Avalon, AXI, and APB protocols.

Platform Designer 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, Platform Designer 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 Platform Designer interconnect provides any necessary adaptation between them. In the path between master and slaves, Platform Designer 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.
Platform Designer 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. Platform Designer 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:* Since interface connections between AMBA 3 AXI and AMBA 4 AXI declare a fixed set of signals with variable latency, there is no need for adapting between active-low and active-high signaling, burst characteristics, different latencies, or port signatures. Adaptation might be necessary between Avalon interfaces.

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 Platform Designer interconnect to slaves in the Platform Designer system.

The dark blue blocks represent interconnect components. The dark gray boxes indicate items outside of the Platform Designer system and the Intel Quartus Prime software design, and show how to export component interfaces and how to connect these interfaces to external devices.
3.1.1. Platform Designer Packet Format

The Platform Designer packet format supports Avalon, AXI, and APB transactions. Memory-mapped transactions between masters and slaves are encapsulated in Platform Designer packets. For Avalon systems without AXI or APB interfaces, some fields are ignored or removed.
3.1.1.1. Fields in the Platform Designer Packet Format

The fields of the Platform Designer packet format are of variable length to minimize resource usage. However, if most components in a design have a single data width, for example 32-bits, and a single component has a data width of 64-bits, Platform Designer inserts a width adapter to accommodate 64-bit transfers.

Table 16. Platform Designer Packet Format for Memory-Mapped Master and Slave Interfaces

<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. 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 &quot;address&quot; 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>
</tbody>
</table>
| Byteenable               | Specifies which symbols are valid. AXI can issue or accept any byteenable pattern. For compatibility with Avalon, Intel recommends that you use the following legal values for 32-bit data transactions between Avalon masters and slaves:  
  • 1111—Writes full 32 bits  
  • 0011—Writes lower 2 bytes  
  • 1100—Writes upper 2 bytes  
  • 0001—Writes byte 0 only  
  • 0010—Writes byte 1 only  
  • 0100—Writes byte 2 only  
  • 1000—Writes byte 3 only |
| Source_ID                | The ID of the master or slave that initiated the command or response.       |
| Destination_ID           | The ID of the master or slave to which the command or response is directed. |
| Response                 | Carries the AXI response signals.                                          |
| Thread ID                | Carries the AXI transaction ID values.                                      |
| Byte count               | The number of bytes remaining in the transaction, including this beat. Number of bytes requested by the packet. |

continued...
### Command Description

**Burstwrap**

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:

- **Variable wrap**—Variable wrap bursts can wrap at any integer power of 2 value. When the burst reaches the wrap boundary, it wraps back to the previous burst boundary so that only the low order bits are used for addressing. For example, a burst starting at address 0x1C, with a burst wrap boundary of 32 bytes and a burst size of 20 bytes, would write to addresses 0x1C, 0x0, 0x4, 0x8, and 0xC.

- **For a burst wrap boundary of size \(<m>\), Burstwrap = \(<m> - 1\), or for this case Burstwrap = \((32 - 1) = 31\) which is \(2^5 - 1\).

- **For AXI masters**, the burstwrap boundary value (\(m\)) is based on the different AXBURST:
  - Burstwrap set to all 1’s. For example, for a 6-bit burstwrap, burstwrap is 6'b111111.
  - For WRAP bursts, burstwrap = AXLEN * size – 1.
  - For FIXED bursts, burstwrap = size – 1.
  - Sequential bursts increment the address for each transfer in the burst. For sequential bursts, the Burstwrap field is set to all 1s. For example, with a 6-bit Burstwrap field, the value for a sequential burst is 6'b111111 or 63 which is \(2^6 - 1\).

For Avalon masters, Platform Designer adaptation logic sets a hardwired value for the burstwrap field, according to the declared master burst properties. For example, for a master that declares sequential bursting, the burstwrap field is set to ones. Similarly, masters that declare burst have their burstwrap field set to the appropriate constant value.

AXI masters choose their burst type at run-time, depending on the value of the AW or ARBURST signal. The interconnect calculates the burstwrap value at run-time for AXI masters.

### Protection

Access level protection. When the lowest bit is 0, the packet has normal access. When the lowest bit is 1, the packet has privileged access. For Avalon-MM interfaces, this field maps directly to the privileged access signal, which allows a memory-mapped master to write to an on-chip memory ROM instance. The other bits in this field support AXI secure accesses and use the same encoding, as described in the AXI specification.

### QoS

QoS (Quality of Service Signaling) is a 4-bit field that is part of the AMBA 4 AXI interface that carries QoS information for the packet from the AXI master to the AXI slave. Transactions from AMBA 3 AXI 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.

### Data sideband

Carries data sideband signals for the packet. On a write command, the data sideband directly maps to WUSER. On a read response, the data sideband directly maps to RUSER. On a write response, the data sideband directly maps to BUSER.

### 3.1.1.2. Transaction Types for Memory-Mapped Interfaces

**Table 17. 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 the read command can be expressed in a single cycle (all byteenable 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>
3.1.1.3. Platform Designer Transformations

The memory-mapped master and slave components connect to network interface modules that encapsulate the transaction in Avalon-ST packets. The memory-mapped interfaces have no information about the encapsulation or the function of the layer transporting the packets. The interfaces operate in accordance with memory-mapped protocol and use the read and write signals and transfers.

Figure 78. Transformation when Generating a System with Memory-Mapped and Slave Components

Platform Designer components that implement the blocks appear shaded.

Related Information
- Master Network Interfaces on page 131
- Slave Network Interfaces on page 134

3.1.2. 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.

3.1.2.1. Using One Domain with Width Adaptation

When one of the masters in a system connects to all the slaves, Platform Designer 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 79. **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.
3.1.2.2. Using Two Separate Domains

**Figure 80. Two Separate Domains**

In this system example, Platform Designer 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, Platform Designer 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.

3.1.3. Master Network Interfaces

**Figure 81. 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.
An AMBA 4 AXI 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**
- Avalon Interface Specifications
- Creating a System with Platform Designer on page 10

### 3.1.3.1. Avalon-MM Master Agent

The Avalon-MM Master Agent translates Avalon-MM master transactions into Platform Designer command packets and translates the Platform Designer Avalon-MM slave response packets into Avalon-MM responses.

### 3.1.3.2. 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 Platform Designer.

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

### 3.1.3.3. AXI Master Agent

An AXI Master Agent accepts AXI commands and produces Platform Designer command packets. It also accepts Platform Designer 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 Platform Designer Packet Format.

**Related Information**
Fields in the Platform Designer Packet Format on page 127

### 3.1.3.4. AXI Translator

AMBA 4 AXI allows omitting signals from interfaces. The translator bridges between these "incomplete" AMBA 4 AXI interfaces and the "complete" AMBA 4 AXI interface on the network interfaces.

**Attention:** If an Avalon or AMBA 4 AXI slave is connected to a master without response ports, the interconnect could ignore transaction responses such as SLAVEERROR or DECODEERROR. This situation could lead to returning invalid data to the master.

The AXI translator is inserted for both AMBA 4 AXI 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 AMBA 3 AXI master connects to an AMBA 4 AXI slave in 1x1 systems.

**Related Information**
Arm AMBA Protocol Specifications

### 3.1.3.5. APB Master Agent

An APB master agent accepts APB commands and produces or generates Platform Designer command packets. It also converts Platform Designer response packets to APB responses.

### 3.1.3.6. APB Slave Agent

An APB slave agent issues resulting transaction to the APB interface. It also accepts creates Platform Designer response packets.

### 3.1.3.7. 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 (Intel SoC's Hard Processor System).
3.1.3.8. AHB Slave Agent

The Platform Designer interconnect supports non-bursting Advanced High-performance Bus (AHB) slave interfaces.

3.1.3.9. 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.

3.1.3.10. 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.

3.1.4. Slave Network Interfaces

3.1.4.1. Avalon-MM Slave Translator

The Avalon-MM Slave Translator converts the Avalon-MM slave interface to a simplified representation that the Platform Designer network can use.

Figure 83. Avalon-MM Slave Network Interface

An Avalon-MM 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 134

3.1.4.2. AXI Translator

AMBA 4 AXI allows omitting signals from interfaces. The translator bridges between these "incomplete" AMBA 4 AXI interfaces and the "complete" AMBA 4 AXI interface on the network interfaces.

Figure 84. AXI Slave Network Interface

An AMBA 4 AXI slave supports up to 256 beat INCR bursts, QoS signals, and data sideband signals.

The AXI translator is inserted for both AMBA 4 AXI 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 AMBA 3 AXI master connects to an AMBA 4 AXI slave in 1x1 systems.

3.1.4.3. 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. Platform Designer 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 85. **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. Platform Designer interconnect can force a master to wait for the wait state needs of a slave; for example, arbitration logic in a multi-master system. Platform Designer generates wait state insertion logic based on the properties of all slaves in the system.

### 3.1.4.4. 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.

### 3.1.4.5. AXI Slave Agent

An AXI Slave Agent works like a reverse master agent. The AXI Slave Agent accepts Platform Designer command packets to create AXI commands, and accepts AXI responses to create Platform Designer response packets. This component has separate packet channels for read commands, write commands, read responses, and write responses.

### 3.1.5. Arbitration

When multiple masters contend for access to a slave, Platform Designer 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 Platform Designer GUI.

#### 3.1.5.1. Round-Robin Arbitration

When multiple masters contend for access to a slave, Platform Designer 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 given 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**.
3.1.5.1.1. 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 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. Platform Designer includes only requesting masters in the arbitration for each slave transaction.

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 Platform Designer 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.

3.1.5.2.1. Designate a Platform Designer Slave to Use Fixed Priority Arbitration

You can designate any slave in your Platform Designer 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 Platform Designer, 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.
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.
9. Right click the designated fixed priority slave and uncheck Show Arbitration Shares to return to the connection circles.

3.1.5.2.2. 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, Platform Designer 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.
3.1.6. Memory-Mapped Arbiter

The input to the Memory-Mapped Arbiter is the command packet for all masters requesting access to a specific 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.
Figure 90. Arbitration Logic

In this example, four Avalon-MM masters connect to four Avalon-MM slaves. In each cycle, an arbiter positioned in front of each Avalon-MM slave selects among the requesting Avalon-MM masters. Logic included in the Avalon-ST Command Network.

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

Note: You can use the Memory-Mapped Arbiter for both round-robin and fixed priority arbitration.
### 3.1.7. 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. Platform Designer generates separate datapath multiplexing logic for every master in the system (`readdata`), and for every slave in the system (`writedata`). Platform Designer does not generate multiplexing logic if it is not needed.

#### Figure 91. Datapath Multiplexing Logic for One Master and Two Slaves

![Datapath Multiplexing Logic Diagram]

### 3.1.8. Width Adaptation

Platform Designer 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.

#### 3.1.8.1. Memory-Mapped Width Adapter

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

The memory-mapped width adapter accepts packets on its sink interface with one data width and produces output packets on its source interface with a different data width. The ratio of the narrow data width must be a power of two, such as 1:4, 1:8, and 1:16. The ratio of the wider data width to the narrower width must also be a power of two, such as 4:1, 8:1, and 16:1. These output packets may have a different size if the input size exceeds the output data bus width, or if data packing is enabled.

When the width adapter converts from narrow data to wide data, each input beat’s data and byte enables are copied to the appropriate segment of the wider output data and byte enables signals.
Figure 92. **Width Adapter Timing for a 4:1 Adapter**

This adapter assumes that the field ordering of the input and output packets is the same, with the only difference being the width of the data and accompanying byte enable fields. When the width adapter converts from wide data to narrow data, the narrower data is transmitted over several beats. The first output beat contains the lowest addressed segment of the input data and byte enables.

<table>
<thead>
<tr>
<th>Adapter Input</th>
<th>Adapter Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr_in[7:0]</td>
<td>08</td>
</tr>
<tr>
<td>bytenable_in[3:0]</td>
<td>C</td>
</tr>
<tr>
<td>wide_data[31:0]</td>
<td>AABBCCDD</td>
</tr>
<tr>
<td>addr_out[7:0]</td>
<td>08 09 0A 08</td>
</tr>
<tr>
<td>bytenable_out[3:0]</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>narrow_data[7:0]</td>
<td>DD CC BB AA</td>
</tr>
</tbody>
</table>

3.1.8.1.1. 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 18. **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. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths. For example, for a 2:1 downsizing ratio, an INCR9 burst is converted into INCR16 + INCR2 bursts. This is true if the maximum burstcount a slave can accept is 16, which is the case for AMBA 3 AXI slaves. Avalon slaves have a maximum burstcount of 64.</td>
</tr>
<tr>
<td>Wrapping</td>
<td>If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted to a wrapping burst with a larger length, with a size equal to the output width. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths; respecting wrap boundaries. For example, for a 2:1 downsizing ratio, a WRAP16 burst is converted into two or three INCR bursts, depending on the address.</td>
</tr>
<tr>
<td>Fixed</td>
<td>If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted into repeated sequential bursts over the same addresses. For example, for a 2:1 downsizing ratio, a FIXED single burst is converted into an INCR2 burst.</td>
</tr>
</tbody>
</table>
3.1.8.1.2. AXI Narrow-to-Wide Adaptation

Table 19. AXI Narrow-to-Wide Adaptation (Upsizing)

<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>

3.1.9. Burst Adapter

Platform Designer 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 specific 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 AMBA 4 AXI slaves, Platform Designer 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 AMBA 3 AXI slaves.

Avalon-MM masters always issue addresses that are aligned to the size of the transfer. However, when Platform Designer 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 AMBA 4 AXI to AMBA 3 AXI connections, Platform Designer follows an AMBA 4 AXI 256 burst length to AMBA 3 AXI 16 burst length.

3.1.9.1. Burst Adapter Implementation Options

Platform Designer 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 can 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 specific 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 Platform Designer.

Related Information
Creating a System with Platform Designer on page 10

3.1.9.2. Burst Adaptation: AXI to Avalon

Table 20. 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 $\text{slave_max_burst_length}$ are converted to multiple sequential bursts of a length less than or equal to the $\text{slave_max_burst_length}$. Otherwise, the burst is unconverted. For example, for an Avalon slave with a maximum burst length of 4, an $\text{INCR7}$ burst is converted to $\text{INCR4 + INCR3}$.&lt;br&gt;&lt;br&gt;<strong>Wrapping Slave</strong>&lt;br&gt;Bursts that exceed the $\text{slave_max_burst_length}$ are converted to multiple sequential bursts of length less than or equal to the $\text{slave_max_burst_length}$. Bursts that exceed the wrapping boundary are converted to multiple sequential bursts that respect the slave’s wrapping boundary.</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 $\text{max_burst_length}$ and respect the transaction’s wrapping boundary&lt;br&gt;&lt;br&gt;<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>
3.1.9.3. Burst Adaptation: Avalon to AXI

Table 21. Burst Adaptation: Avalon to AXI

<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 INCR bursts of a length less than or equal to 16. Bursts of length less than or equal to 16 are not converted.</td>
</tr>
<tr>
<td>Wrapping</td>
<td>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.</td>
</tr>
<tr>
<td>GENERIC_CONVERTER</td>
<td>Controls all burst conversions with a single converter that adapts all incoming burst types, resulting in an adapter that has smaller area, but lower fMAX.</td>
</tr>
</tbody>
</table>

3.1.10. Waitrequest Allowance Adapter

The Waitrequest Allowance Adapter allows a connection between a master and a slave interface with different waitrequestAllowance properties.

The Waitrequest Allowance adapter provides the following features:

- The adapter is used in the memory-mapped domain and operates with signals on the memory-mapped interface.
- Signal widths and all properties other than waitrequestAllowance are identical on master and slave interfaces.
- The adapter does not modify any command properties such as data width, burst type, or burst count.
- The adapter is inserted by the Platform Designer interconnect software when a master and slave with different waitrequestAllowance property are connected.

When the slave has a waitrequestAllowance = n the master must deassert read or write signals after <n> transfers when waitrequest is asserted.

Table 22. Interconnect Scenarios Requiring waitrequestAllowance

<table>
<thead>
<tr>
<th>Master (m) / Slave (n) waitrequestAllowance</th>
<th>Adaptation Required</th>
<th>Description</th>
<th>Adapter Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = n</td>
<td>No</td>
<td>The master waitrequestAllowance is equal to the slave's waitrequestAllowance.</td>
<td>All signals are passed through.</td>
</tr>
<tr>
<td>m = 0; n &gt; 0</td>
<td>Yes</td>
<td>The master cannot send when waitrequest=1, but holds the value on the bus. This would result in the slave receiving multiple copies. Requires adaptation to prevent.</td>
<td>The adapter deasserts valid when input waitrequest is asserted.</td>
</tr>
<tr>
<td>m &lt; n; m != 0</td>
<td>No</td>
<td>The master can send &lt;m&gt; transfers after waitrequest is asserted. The slave receives fewer than &lt;n&gt; transfers, which is acceptable.</td>
<td>All signals are passed through.</td>
</tr>
<tr>
<td>m &gt; n; n = 0</td>
<td>Yes</td>
<td>The slave cannot accept transfers when waitrequest is asserted. Transfers sent when waitrequest=1 can be lost.</td>
<td>If the input waitrequest is asserted, the adapter buffers the input data.</td>
</tr>
</tbody>
</table>

continued...
Table 23.  Response Support for Mismatched Master and Slave

<table>
<thead>
<tr>
<th>Slave with Response</th>
<th>Slave Without Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Master with Response</strong></td>
<td>Interconnect delivers response from the slave to the master. Response merging or duplication may be necessary for bus sizing.</td>
</tr>
<tr>
<td><strong>Master without Response</strong></td>
<td>Master ignores responses from the slave</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 Platform Designer system security, refer to Manage System Security. For information about specifying a default slave, refer to Error Response Slave in Platform Designer System Design Components.
**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, Platform Designer interconnect generates an OKAY response on behalf of the Avalon-MM slave.

**Related Information**
- Master Network Interfaces on page 131
- Error Response Slave on page 233
- Error Correction Coding (ECC) in Platform Designer Interconnect on page 184

### 3.1.12. Platform Designer 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 93. Address Decoding for One Master and Two Slaves**

In this example, Platform Designer generates separate address decoding logic for each master in a system. The address decoding logic processes the difference between the master address width (<M>) and the individual slave address widths (<S> and <T>). The address decoding logic also maps only the necessary master address bits to access words in each slave’s address space.

Platform Designer 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 Platform Designer interconnect to allow the address decoding logic to be efficient, and to achieve the best possible f\(_{\text{MAX}}\).

### 3.2. Avalon Streaming Interfaces

High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. Streaming interfaces can also use memory-mapped connection interfaces to provide an access point for control. In contrast to the memory-mapped interconnect, the Avalon-ST interconnect always creates a point-to-point connection between a single data source and data sink.
Figure 94. **Memory-Mapped and Avalon-ST Interfaces**

In this example, there are the following connection pairs:
- Data source in the Rx Interface transfers data to the data sink in the FIFO.
- Data source in the FIFO transfers data to the Tx Interface data sink.

The memory-mapped interface allows a processor to access the data source, FIFO, or data sink to provide system control. If your source and sink interfaces have different formats, for example, a 32-bit source and an 8-bit sink, Platform Designer automatically inserts the necessary adapters. You can view the adapters on the System Contents tab by clicking System ➤ Show System with Platform Designer Interconnect.

![Diagram of Memory-Mapped and Avalon-ST Interfaces](image)

Figure 95. **Avalon-ST Connection Between the Source and Sink**

This source-sink pair includes only the data signal. The sink must be able to receive data as soon as the source interface comes out of reset.

![Diagram of Avalon-ST Connection Between the Source and Sink](image)
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 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 Platform Designer generation creates interconnect with a structure similar to a network topology, as described in Platform Designer Transformations. The following sections introduce the Avalon-ST components.

**Related Information**
Platform Designer Transformations on page 129

### 3.2.1. Avalon-ST Adapters

Platform Designer 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, Platform Designer inserts the appropriate adapter type to connect the mismatched interfaces.

After generation, you can view the inserted adapters selecting System ➤ Show System With Platform Designer Interconnect. For each mismatched source-sink pair, Platform Designer 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, Platform Designer does not insert adapters and reports the mismatched interface with validation error message.
The auto-inserted adapters feature does not work for video IP core connections.

### 3.2.1.1. 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, Platform Designer instantiates the necessary adapter logic (channel, error, data format, or timing) as hierarchal sub-components.

#### 3.2.1.1.1. Avalon-ST Adapter Parameters Common to Source and Sink Interfaces

**Table 24. 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>

#### 3.2.1.1.2. Avalon-ST Adapter Upstream Source Interface Parameters

**Table 25. 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>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>
</tbody>
</table>
### 3.2.1.1.3. Avalon-ST Adapter Downstream Sink Interface Parameters

#### Table 26. Avalon-ST Adapter Downstream Sink Interface Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>Sink Ready Latency</td>
<td>Specifies what ready latency to expect from the source interface connected to the sink interface.</td>
</tr>
</tbody>
</table>

### 3.2.1.2. Channel Adapter

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

#### Table 27. 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>Platform Designer 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>Platform Designer 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. Platform Designer 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>
3.2.1.2.1. Avalon-ST Channel Adapter Input Interface Parameters

Table 28. 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>

3.2.1.2.2. Avalon-ST Channel Adapter Output Interface Parameters

Table 29. 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>

3.2.1.2.3. Avalon-ST Channel Adapter Common to Input and Output Interface Parameters

Table 30. 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>

3.2.1.3. 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 31. Data Format Adapter Adaptations

<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>
<tr>
<td>The source and sink have a different number of symbols per beat.</td>
<td>The adapter converts the source’s width to the sink’s width.</td>
</tr>
</tbody>
</table>

continued...
**Condition** | **Description of Adapter Logic**
---|---
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. Platform Designer cannot make the connection.

**Figure 97. 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.

**3.2.1.3.1. Avalon-ST Data Format Adapter Input Interface Parameters**

**Table 32. 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>

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

**Table 33. 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>
3.2.1.3.3. Avalon-ST Data Format Adapter Common to Input and Output Interface Parameters

Table 34. 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, Platform Designer uses startofpacket, endofpacket, and 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>

3.2.1.4. 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 source and sink can 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. Platform Designer assigns the bit-wise ORing of all input error bits that are unmatched, to the output interface error bits set with the other descriptor.

3.2.1.4.1. Avalon-ST Error Adapter Input Interface Parameters

Table 35. 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>
3.2.1.4.2. Avalon-ST Error Adapter Output Interface Parameters

Table 36. 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>

3.2.1.4.3. Avalon-ST Error Adapter Common to Input and Output Interface Parameters

Table 37. 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 startofpacket, 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>

3.2.1.5. Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO buffer between the source and sink to buffer data or pipeline stages to delay the back-pressure signals. You can also use the timing adapter to connect interfaces that support the ready signal, and those that do not. The timing adapter treats all signals other than the ready and valid signals as payload, and simply drives them from the source to the sink.
Table 38. Timing Adapter Adaptations

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

3.2.1.5.1. Avalon-ST Timing Adapter Input Interface Parameters

Table 39. 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 ready signal</td>
<td>Indicates whether a ready 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>

3.2.1.5.2. Avalon-ST Timing Adapter Output Interface Parameters

Table 40. Avalon-ST Timing Adapter 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>Indicates whether a ready 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>

3.2.1.5.3. Avalon-ST Timing Adapter Common to Input and Output Interface Parameters

Table 41. 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 startofpacket, 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>
<tr>
<td>Data Symbols Per Beat</td>
<td>Number of symbols per active transfer.</td>
</tr>
</tbody>
</table>

continued...
<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>
<tr>
<td>Error Signal Width (bits)</td>
<td>Width of the output error signal in bits.</td>
</tr>
<tr>
<td>Error Signal Description</td>
<td>A list of strings that describes errors.</td>
</tr>
</tbody>
</table>

### 3.3. 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. Platform Designer does not insert interrupt synchronizers for such receivers.

For clock crossing adaption on interrupts, Platform Designer 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). Platform Designer inserts the adapter if there is any kind of mismatch between the start and end points. Platform Designer does not insert the adapter if the interrupt receiver does not have an associated clock.

**Related Information**

**IRQ Mapper** on page 160

### 3.3.1. Individual Requests IRQ Scheme

In the individual requests IRQ scheme, Platform Designer interconnect passes IRQs directly from the sender to the receiver, without making assumptions about IRQ priority. If multiple senders assert their IRQs simultaneously, the receiver logic determines which IRQ has highest priority, and then responds appropriately.
### Interrupt Controller Mapping IRQs

Using individual requests, the interrupt controller can process up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31:0]` to the receiver, and maps slave IRQ signals to the bits of `irq[31:0]`. Any unassigned bits of `irq[31:0]` are disabled.

![Interrupt Controller Diagram]

### 3.3.2. Assigning IRQs in Platform Designer

You assign IRQ connections on the **System Contents** tab of Platform Designer. 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. Platform Designer uses the following three components to implement interrupt handling: IRQ Bridge, IRQ Mapper, and IRQ Clock Crosser.

#### 3.3.2.1. IRQ Bridge

The IRQ Bridge allows you to route interrupt wires between Platform Designer subsystems.
**Figure 99. Platform Designer 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.

![Diagram of Platform Designer IRQ Bridge Application](image)

**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 Platform Designer interconnect generation. Platform Designer 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. Platform Designer 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.

### 3.3.2.2 IRQ Mapper

Platform Designer inserts the IRQ Mapper automatically during generation. The IRQ Mapper converts individual interrupt wires to a bus, and then maps the appropriate IRQ priority number onto the bus.

By default, the interrupt sender connected to the `receiver0` interface of the IRQ mapper is the highest priority, and sequential receivers are successively lower priority. You can modify the interrupt priority of each IRQ wire by modifying the IRQ priority number in Platform Designer under the `IRQ` column. The modified priority is reflected in the `IRQ_MAP` parameter for the auto-inserted IRQ Mapper.
Figure 100. IRQ Column in Platform Designer

Circled in the IRQ column are the default interrupt priorities allocated for the CPU subsystem.

### Related Information

*IRQ Bridge* on page 159

#### 3.3.2.3. 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. Platform Designer automatically inserts this component when it is required.

#### 3.4. 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**

### 3.4.1. (High Speed Serial Interface) HSSI Clock Interfaces

You can use HSSI Serial Clock and HSSI Bonded Clock interfaces in Platform Designer to enable high speed serial connectivity between clocks that are used by certain IP protocols.

#### 3.4.1.1. 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.

#### 3.4.1.1.1. 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:

```tcl
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 42. 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 43. 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>

3.4.1.1.2. HSSI Serial Clock Sink

The HSSI Serial Clock interface includes a sink in the **End** direction.

You can instantiate the HSSI Serial Clock Sink interface in the `_hw.tcl` file as:

```
add_interface <name> hssi_serial_clock end
```

You can connect the HSSI Serial Clock Sink interface to a single HSSI Serial Clock Source interface; you cannot connect it to multiple sources. This Interface has a single **clk** port role limited to a 1 bit width, and a **clockRate** parameter, which is the frequency of the clock driven by the HSSI Serial Clock Source interface.

An unconnected and unexported HSSI Serial Sink is invalid and generates error messages.

Table 44. 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 45. 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>

3.4.1.1.3. 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 the following criteria are satisfied. If the following criteria are not satisfied, Platform Designer 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.**
3.4.1.4. HSSI Serial Clock Example

Example 5. 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 TOPLEVEL "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. 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
```

3.4.1.2. HSSI Bonded Clock Interface

You can connect the HSSI Bonded Clock interface only with similar type of interfaces, for example, you can connect a HSSI Bonded Clock Source interface to a HSSI Bonded Clock Sink interface.
3.4.1.2.1. 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>
<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 47. 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>serializationFactor</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>

3.4.1.2.2. 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 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 48. 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 49. 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>

3.4.1.2.3. 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 the following criteria are satisfied. If the following criteria are not satisfied, Platform Designer 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, Platform Designer generates a warning if the serializationFactor of HSSI Bonded Clock Source is not same as the serializationFactor of the HSSI Bonded Clock Sink.

3.4.1.2.4. HSSI Bonded Clock Example

Example 7. 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_filesynthesis QUARTUS_SYNTH generate
add_filesetverilog_simulation SIM_VERILOG generate
set_fileset_property synthesis TOP_LEVEL "hssi_bonded_component"
set_fileset_propertyverilog_simulation TOP_LEVEL \
```
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 8. HSII Bonded Clock Instantiated in a Composed Component**

```plaintext
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 \n    hssi_bonded_clock
# adding connection from source of myinst2 to sink of myinst1
add_connection myinst2.my_clock_start myinst2.my_clock_end \n    hssi_bonded_clock
```

## 3.5. 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.

**Related Information**

[Specifying Interconnect Requirements](#) on page 35
3.5.1. Single Global Reset Signal Implemented by Platform Designer

When you select System ➤ Create Global Reset Network, the Platform Designer interconnect creates a global reset bus. All 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 Platform Designer interconnect inserts the system-wide reset under the following conditions:

- The global reset input to the Platform Designer system is asserted.
- Any component asserts its reset request signal.

3.5.2. Reset Controller

Platform Designer 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.

Platform Designer 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

3.5.3. Reset Bridge

The Reset Bridge allows you to use a reset signal in two or more subsystems of your Platform Designer 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:* Platform Designer 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.

### 3.5.4. Reset Sequencer

The Reset Sequencer allows you to control the assertion and deassertion sequence for Platform Designer system resets.

The Parameter Editor displays the expected assertion and deassertion sequences based on the current settings. You can connect multiple reset sources to the reset sequencer, and then connect the outputs of the Reset Sequencer to components in the system.
3.5.4.1. Reset Sequencer Parameters

Table 50. Reset Sequencer Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of reset outputs</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>Number of reset inputs</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>Minimum reset assertion time</td>
<td>Specifies the minimum assertion cycles between the assertion of the last sequenced reset, and the deassertion of the first sequenced reset. The range is 0 to 1023.</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>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSRT Seq#</td>
<td>Determines the reset order of reset deassertion. Enter the values 1, 2, 3, etc. to specify the required non-overlapping deassertion 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 deasserting or deglitching the reset. If the USE_DSRT_QUAL parameter is set to 0, specifies the number of cycles to wait before deasserting the reset. If USE_DSRTQUAL is set to 1, specifies the number of cycles to deglitch the input reset_dsrt_qual signal. This value determines either the DSRT_DELAY, or the DSRT_QUALCNT value in the component HDL, depending on the USE_DSRT_QUAL parameter setting. The range is 0 to 1023.</td>
</tr>
<tr>
<td>USE_DSRT_QUAL</td>
<td>If you set USE_DSRT_QUAL to 1, the deassertion 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 deassertion, set this parameter to 0.</td>
</tr>
</tbody>
</table>

### 3.5.4.2. Reset Sequencer Timing Diagrams

**Figure 102. Basic Sequencing**

![Basic Sequencing Diagram](image1)

**Figure 103. Sequencing with USE_DSRT_QUAL Set**

![Sequencing with USE_DSRT_QUAL Set Diagram](image2)
3.5.4.3. Reset Sequencer CSR Registers

The Reset Sequencer's CSR registers provide the following functionality:

- **Support reset logging**
  - Ability to identify which reset is asserted.
  - Ability to determine whether any reset is currently active.

- **Support software triggered resets**
  - Ability to generate reset by writing to the register.
  - Ability to disable assertion or deassertion sequence.

- **Support software sequenced reset**
  - Ability for the software to fully control the assertion/deassertion sequence by writing to registers and stepping through the sequence.

- **Support reset override**
  - Ability to assert a specific component reset through software.

### Table 51. Reset Sequencer CSR Register Map

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Width</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register</td>
<td>0x00</td>
<td>32</td>
<td>0x0</td>
<td>The Status register indicates which sources are allowed to cause a reset.</td>
</tr>
<tr>
<td>Interrupt Enable Register</td>
<td>0x04</td>
<td>32</td>
<td>0x0</td>
<td>The Interrupt Enable register bits enable events triggering the IRQ of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reset sequencer.</td>
</tr>
<tr>
<td>Control Register</td>
<td>0x08</td>
<td>32</td>
<td>0x0</td>
<td>The Control register allows you to control the Reset Sequencer.</td>
</tr>
<tr>
<td>Software Sequenced Reset</td>
<td>0x0C</td>
<td>32</td>
<td>0x3FF</td>
<td>You can program the Software Sequenced Reset Assert control register to</td>
</tr>
<tr>
<td>Assert Control Register</td>
<td></td>
<td></td>
<td></td>
<td>control the reset assertion sequence.</td>
</tr>
<tr>
<td>Software Sequenced Reset</td>
<td>0x10</td>
<td>32</td>
<td>0x3FF</td>
<td>You can program the Software Sequenced Reset Deassert control register to</td>
</tr>
<tr>
<td>Deassert Control Register</td>
<td></td>
<td></td>
<td></td>
<td>control the reset deassertion sequence.</td>
</tr>
<tr>
<td>Software Direct Controlled</td>
<td>0x14</td>
<td>32</td>
<td>0x0</td>
<td>You can write a bit to 1 to assert the reset_outN signal, and to 0 to</td>
</tr>
<tr>
<td>Resets</td>
<td></td>
<td></td>
<td></td>
<td>deassert the reset_outN signal.</td>
</tr>
<tr>
<td>Software Reset Masking</td>
<td>0x18</td>
<td>32</td>
<td>0x0</td>
<td>Masking off (writing 1) to a reset_outN &quot;Reset Mask Enable&quot; signal prevents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the corresponding reset from being asserted. Writing a bit to 0 to a reset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mask enable signal allows assertion of reset_outN.</td>
</tr>
</tbody>
</table>

3.5.4.3.1. Reset Sequencer Status Register

The Status register indicates which sources are allowed to cause a reset.

You can clear bits by writing 1 to the bit location. The Reset Sequencer ignores attempts to write 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 52. Values for the Status Register at Offset 0x00

<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 deassertion).</td>
</tr>
<tr>
<td>30</td>
<td>RW1C</td>
<td>0</td>
<td>Reset Asserted and waiting for SW to proceed—Set when there is an active reset assertion, and the next sequence is waiting for the software to proceed. Only valid when the Enable SW sequenced reset assert option is turned on.</td>
</tr>
<tr>
<td>29</td>
<td>RW1C</td>
<td>0</td>
<td>Reset Deasserted and waiting for SW to proceed—Set when there is an active reset deassertion, and the next sequence is waiting for the software to proceed. Only valid when the Enable SW sequenced reset deassert option is turned on.</td>
</tr>
<tr>
<td>28:26</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25:16</td>
<td>RW1C</td>
<td>0</td>
<td>Reset deassertion input qualification signal reset_dsrt_qual [9:0] status—Indicates that the reset deassertion'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>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in9 was triggered—Indicates that reset_in9 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>10</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in8 was triggered—Indicates that reset_in8 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>9</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in7 was triggered—Indicates that reset_in7 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>8</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in6 was triggered—Indicates that reset_in6 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>7</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in5 was triggered—Indicates that reset_in5 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>6</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in4 was triggered—Indicates that reset_in4 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>5</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in3 was triggered—Indicates that reset_in3 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>4</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in2 was triggered—Indicates that reset_in2 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>3</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in1 was triggered—Indicates that reset_in1 triggered the reset. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>2</td>
<td>RW1C</td>
<td>0</td>
<td>reset_in0 was triggered—Indicates that reset_in0 triggered. Software clears this bit by writing 1 to this location.</td>
</tr>
<tr>
<td>1</td>
<td>RW1C</td>
<td>0</td>
<td>Software-triggered reset—Indicates that the software-triggered reset is set by the software, and triggering a reset.</td>
</tr>
<tr>
<td>0</td>
<td>RW1C</td>
<td>0</td>
<td>Power-on-reset was triggered—Asserted whenever the reset to the sequencer is triggered. This bit is NOT reset when sequencer is reset. Software clears this bit by writing 1 to this location.</td>
</tr>
</tbody>
</table>

### Related Information

**Reset Sequencer CSR Registers** on page 172
3.5.4.3.2. Reset Sequencer Interrupt Enable Register

The Interrupt Enable register bits enable events triggering the IRQ of the reset sequencer.

Table 53. 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></td>
<td>Reserved.</td>
<td>Reserved.</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 Deasserted 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 deassertion sequence.</td>
</tr>
<tr>
<td>28:26</td>
<td></td>
<td>Reserved.</td>
<td>Reserved.</td>
</tr>
<tr>
<td>25:16</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on Reset deassertion 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></td>
<td>Reserved.</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>1</td>
<td>RW</td>
<td>0</td>
<td>Interrupt on Software triggered reset Enable—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>Interrupt on Power-On-Reset Enable—When set, the IRQ is set when the power-on-reset status bit is set.</td>
</tr>
</tbody>
</table>

Related Information
Reset Sequencer CSR Registers on page 172
3.5.4.3.3. Reset Sequencer Control Register

The Control register allows you to control the Reset Sequencer.

### Table 54. 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></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td>0</td>
<td>Enable SW sequenced reset assert—Enable a software sequenced reset assert sequence. Timer delays and input qualification are ignored, and only the software can sequence the assert.</td>
</tr>
<tr>
<td>1</td>
<td>RW</td>
<td>0</td>
<td>Enable SW sequenced reset deassert—Enable a software sequenced reset deassert sequence. Timer delays and input qualification are ignored, and only the software can sequence the deassert.</td>
</tr>
<tr>
<td>0</td>
<td>WO</td>
<td>0</td>
<td>Initiate Reset Sequence—To trigger the hardware sequenced warm reset, the Reset Sequencer writes this bit to 1 a single time. The Reset Sequencer verifies that Reset Active is 0 before setting this bit, and always reads the value 0. To monitor this sequence, verify that Reset Active is asserted, and then subsequently deasserted.</td>
</tr>
</tbody>
</table>

**Related Information**

Reset Sequencer CSR Registers on page 172

3.5.4.3.4. Reset Sequencer Software Sequenced Reset Assert Control Register

You can program the Software Sequenced Reset Assert control register to control the reset assertion sequence.

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 55. Values for the Reset Sequencer Software Sequenced Reset Assert Control 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></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>RW</td>
<td>0x3FF</td>
<td>Per-reset SW sequenced reset assert enable—This is a per-bit enable for SW sequenced reset assert. If the register’s bitN is set, the sequencer sets the bit30 of the status register when a resetN is asserted. It then waits for the bit30 of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).</td>
</tr>
</tbody>
</table>

**Related Information**

Reset Sequencer CSR Registers on page 172

3.5.4.3.5. Reset Sequencer Software Sequenced Reset Deassert Control Register

You can program the Software Sequenced Reset Deassert register to control the reset deassertion sequence.
When the corresponding enable bit is set, the sequencer stops when the desired reset asserts, and then sets the Reset Deasserted and waiting for SW to proceed bit. The Reset Sequencer proceeds only after the Reset Deasserted and waiting for SW to proceed bit is cleared.

**Table 56. Values for the Reset Sequencer Software Sequenced Reset Deassert 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>Reserved.</td>
<td></td>
<td>Per-reset SW sequenced reset deassert enable—This is a per-bit enable for SW-sequenced reset deassert. If bitN of this register is set, the sequencer sets bit29 of the Status Register when a resetN is asserted. It then waits for the bit29 of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).</td>
</tr>
<tr>
<td>9:0</td>
<td>RW</td>
<td>0x3FF</td>
<td>Per-reset SW sequenced reset deassert enable—This is a per-bit enable for SW-sequenced reset deassert. If bitN of this register is set, the sequencer sets bit29 of the Status Register when a resetN is asserted. It then waits for the bit29 of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).</td>
</tr>
</tbody>
</table>

**Related Information**
Reset Sequencer CSR Registers on page 172

### 3.5.4.3.6. Reset Sequencer Software Direct Controlled Resets

You can write a bit to 1 to assert the reset_outN signal, and to 0 to deassert the reset_outN signal.

**Table 57. 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>Reserved.</td>
<td></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>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>Reserved.</td>
<td></td>
<td>reset_outN Reset Overwrite Value—This is a per-bit control of the reset_out bit. The Reset Sequencer can use this to forcefully drive the reset to a specific value. A value of 1 sets the reset_out. A value of 0 clears the reset_out. A write to this register only takes effect if the corresponding trigger bit in this register is set.</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_out bit. The Reset Sequencer can use this to forcefully drive the reset to a specific value. A value of 1 sets the reset_out. A value of 0 clears the reset_out. A write to this register only takes effect if the corresponding trigger bit in this register is set.</td>
</tr>
</tbody>
</table>

**Related Information**
Reset Sequencer CSR Registers on page 172

### 3.5.4.3.7. Reset Sequencer Software Reset Masking

Masking off (writing 1) to a reset_outN "Reset Mask Enable" signal prevents the corresponding reset from being asserted. Writing a bit to 0 to a reset mask enable signal allows assertion of reset_outN.
Table 58. 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>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>RW</td>
<td>0</td>
<td>reset_outN &quot;Reset Mask Enable&quot;—This is a per-bit control to mask off the reset_outN bit. Software Reset Masking prevents the reset bit from being asserted during a reset assertion sequence. If reset_out is already asserted, it does not deassert the reset.</td>
</tr>
</tbody>
</table>

Related Information

Reset Sequencer CSR Registers on page 172
3.5.4.4. Reset Sequencer Software Flows

3.5.4.4.1. Reset Sequencer (Software-Triggered) Flow

Figure 104. Reset Sequencer (Software-Triggered) Flow Diagram

Related Information
- Reset Sequencer Status Register on page 172
- Reset Sequencer Control Register on page 175
3.5.4.4.2. Reset Assert Flow

The following flow sequence occurs for a Reset Assert 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 which reset was triggered.

3.5.4.4.3. Reset Deassert Flow

The following flow sequence occurs for a Reset Deassert Flow:

- When a reset source is deasserted, or when the reset assert sequence has completed without pending resets asserted, the deassertion flow is initiated.
- The IRQ is asserted, if the IRQ is enabled.
- Software reads the Status Register to determine which reset was triggered.

3.5.4.4.4. Reset Assert (Software Sequenced) Flow

**Figure 105. Reset Assert (Software Sequenced) Flow**

**SETUP**

- Software sets Control Register’s Enable SW sequenced reset assert bit
- Software defines which reset sequence it wants to control by setting bits in Software sequenced Reset assert Control register’s Per-reset SW sequenced reset assert enable
- Software sets Interrupt Enable register’s Interrupt on Reset Asserted and waiting for SW to proceed bit

**RUNTIME**

- Hardware sequences a reset until the point where Reset Sequencer must wait for software
- Reset Sequencer asserts an IRQ
- Software waits until reset is asserted by checking if Status Register’s Reset asserted and waiting for SW to proceed bit is set
- Software clears Status Register’s Reset asserted and waiting for SW to proceed bit
- SW writes to SW sequenced Reset Assert control register’s Per-reset SW sequenced reset assert enable
- Reset Sequencer sets IRQ on the next Reset Sequencer trigger point (if any)

Related Information

- Reset Sequencer Control Register on page 175
- Reset Sequencer Software Sequenced Reset Assert Control Register on page 175
- Reset Sequencer Interrupt Enable Register on page 174
- Reset Sequencer Status Register on page 172
3.5.4.5. Reset Deassert (Software Sequenced) Flow

The sequence and flow is similar to the Reset Assert (SW Sequenced) flow, though, this flow uses the reset deassert registers/bits instead of the reset assert registers/bits.

Related Information
Reset Assert (Software Sequenced) Flow on page 179

3.6. 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 Platform Designer 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 Platform Designer, 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 Platform Designer on page 10

3.7. Interconnect Pipelining

Pipeline stages increase a design's $f_{\text{MAX}}$ by reducing the combinational logic depth, at the cost of additional latency and logic.

The Limit interconnect pipeline stages to option in the Interconnect Requirements tab allows you to define the maximum Avalon-ST pipeline stages that Platform Designer can insert during generation. You can specify between 0 to 4 pipeline stages, where 0 means that the interconnect has a combinational datapath. Choosing 3 or 4 pipeline stages may significantly increase the logic utilization of the system.

Platform Designer adds additional latency once on the command path, and once on the response path.
This setting is specific for each Platform Designer system or subsystem, so you can specify a unique interconnect pipeline stage value for each subsystem.

The insertion of pipeline stages depends upon the existence of certain interconnect components. For example, single-slave systems do not have multiplexers; 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 106. **Pipeline Placement in Arbitration Logic**

The example shows the possible placement of up to four potential pipeline stages. Platform Designer places these stages before the input to the demultiplexer, at the output of the multiplexer, between the arbiter and the multiplexer, and at the output of the demultiplexer.

You can manually adjust the number of pipeline stages in the Platform Designer **Memory-Mapped Interconnect** tab.

**Related Information**
- *Previewing the System Interconnect* on page 33
- *Inserting Pipeline Stages to Increase System Frequency* on page 86
3.7.1. Manually Control Pipelining in the Platform Designer Interconnect

The Memory-Mapped Interconnect tab allows you to manipulate pipeline connections in the Platform Designer interconnect.

Consider manually pipelining the interconnect only when changes to the Limit interconnect pipeline stages to option do not improve frequency, and exhausted all other options to achieve timing closure, including the use of a bridge. Perform manual pipelining only in complete systems.

Access the Memory-Mapped Interconnect tab by clicking System ➤ Show System With Platform Designer Interconnect.

1. In the Intel Quartus Prime software, compile the design and run timing analysis.
2. From the timing analysis output, identify the critical path through the interconnect and determine the approximate mid-point.
4. 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.
5. Click Show Pipelinable Locations. Platform Designer display all possible pipeline locations in the interconnect. Right-click the possible pipeline location to insert or remove a pipeline stage.
6. 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.
7. Right-click the location where you want to insert a pipeline, and then click Insert Pipeline.
8. Regenerate the Platform Designer system, recompile the design, and then rerun timing analysis.
9. If necessary, repeat the manual pipelining process again until the design meets the timing requirements.

Manual pipelining has the following limitations:

- If you make changes to the original system’s connectivity after manually pipelining an interconnect, the inserted pipelines may become invalid. Platform Designer displays warning messages when you generate the 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. Do not make changes to the system’s connectivity after manual pipeline insertion.
- Review manually-inserted pipelines when upgrading to newer versions of Platform Designer. Manually-inserted pipelines in one version of Platform Designer may not be valid in a future version.
3.8. Error Correction Coding (ECC) in Platform Designer Interconnect

Error Correction Coding (ECC) logic allows the Platform Designer interconnect to detect and correct errors. Enabling ECC improves data integrity in memory blocks. Platform Designer supports ECC protection for Read Data FIFO (rdata_FIFO) instances only.

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 status, and increase the FIT rate.

Before writing data to the memory device, the ECC logic encodes the data bus with a Hamming code. Then, the ECC logic decodes and performs error checking on the data output.

When you enable ECC, Platform Designer interconnect sends uncorrectable errors arising from memory as DECODEERROR (DECERR) on the Avalon response bus.

![High-Level Implementation of rdata_FIFO with ECC Enabled](image)

**Note:** Enabling ECC logic may increase logic utilization and cause lower $f_{\text{MAX}}$.

**Related Information**
- Read and Write Responses on page 147
- Interconnect Requirements on page 35

3.9. AMBA 3 AXI Protocol Specification Support (version 1.0)

Platform Designer allows memory-mapped connections between AMBA 3 AXI components, AMBA 3 AXI and AMBA 4 AXI components, and AMBA 3 AXI and Avalon interfaces with unique or exceptional support. Refer to the AMBA 3 Protocol Specifications on the ARM website for more information.

**Related Information**
- Arm AMBA Protocol Specifications
- Slave Network Interfaces on page 134

3.9.1. Channels

Platform Designer has the following support and restrictions for AMBA 3 AXI channels.
3.9.1.1. Read and Write Address Channels

Most signals are allowed. However, the following limitations are present in Platform Designer 14.0:

- Supports 64-bit addressing.
- ID width limited to 18-bits.
- HPS-FPGA master interface has a 12-bit ID.

3.9.1.2. Write Data, Write Response, and Read Data Channels

Most signals are allowed. However, the following limitations are present in Platform Designer 14.0:

- Data widths limited to a maximum of 1024-bits
- Limited to a fixed byte width of 8-bits

3.9.1.3. Low Power Channel

Low power extensions are not supported in Platform Designer, version 14.0.

3.9.2. Cache Support

AWCACHE and ARCACHE are passed to an AXI slave unmodified.

3.9.2.1. Bufferable

Platform Designer 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.

3.9.2.2. Cacheable (Modifiable)

Platform Designer 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:

- Platform Designer considers a wide transaction to a narrow slave as modifiable because the size requires reduction.
- Platform Designer 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.
Platform Designer ignores all other bits, for example, read allocate or write allocate because the interconnect does not perform caching. By default, Platform Designer considers Avalon master transactions as non-bufferable and non-cacheable, with the allocate bits tied low.

### 3.9.3. 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 Manage System Security in Creating a System with Platform Designer for more information about secure systems and the TrustZone feature.

**Related Information**

- Configuring Platform Designer System Security on page 46

### 3.9.4. 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.

### 3.9.5. Response Signaling

Full response signaling is supported. Avalon slaves always return OKAY as a response.

### 3.9.6. Ordering Model

Platform Designer interconnect provides responses in the same order as the commands are issued.

To prevent reordering, for slaves that accept reordering depths greater than 0, Platform Designer does not transfer the transaction ID from the master, but provides a constant transaction ID of 0. For slaves that do not reorder, Platform Designer allows the transaction ID to be transferred to the slave. To avoid cyclic dependencies, Platform Designer 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.

#### 3.9.6.1. AXI and Avalon Ordering

There is a potential read-after-write risk when Avalon masters transact to AXI slaves.

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.

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.
3.9.7. Data Buses

Narrow bus transfers are supported. AXI write strobes can have any pattern that is compatible with the address and size information. Intel 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.

3.9.8. 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, Platform Designer modifies unaligned commands from AXI masters to the correct data width. Platform Designer must preserve commands issued by AXI masters when passing the commands to AXI slaves.

*Note:* Unaligned transfers are aligned if downsizing occurs. For example, when downsizing to a bus width narrower than that required by the transaction size, AWSIZE or ARSIZE, the transaction must be modified.

3.9.9. Avalon and AXI Transaction Support

Platform Designer 14.0 supports transactions between Avalon and interfaces, with some limitations.

3.9.9.1. 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.

3.9.9.2. 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.

Platform Designer always assumes that the byteenable is asserted based on the size of the command, not the address of the command. The following scenarios are examples:

- For a 32-bit AXI master that issues a read command with an unaligned address starting at address 0x01, and a burstcount of 2 to a 32-bit Avalon slave, the starting address is: 0x00.
- For a 32-bit AXI master that issues a read command with an unaligned address starting at address 0x01, with 4-bytes to an 8-bit AXI slave, the starting address is: 0x00.
3.10. AMBA 3 APB Protocol Specification Support (version 1.0)

APB (Advanced Peripheral Bus) interface is optimized for minimal power consumption and reduced interface complexity. You can use 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.

Platform Designer allows connections between APB components, and AMBA 3 AXI, AMBA 4 AXI, and Avalon memory-mapped interfaces. The following sections describe unique or exceptional APB support in the Platform Designer software.

Related Information
Arm AMBA Protocol Specifications

3.10.1. Bridges

With APB, you cannot use bridge components that use multiple PSELx in Platform Designer. As a workaround, you can group PSELx, and then send the packet to the slave directly.

Intel recommends as an alternative that you instantiate the APB bridge and all the APB slaves in Platform Designer. 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. Platform Designer 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 Platform Designer. 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 Platform Designer and export APB master to the top-level, and from there connect to APB bus outside of Platform Designer.

3.10.2. 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.

3.10.3. Width Adaptation

Platform Designer 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.
3.10.4. Error Response

Error responses are returned to the master. Platform Designer performs error mapping if the master is an AMBA 3 AXI or AMBA 4 AXI master, for example, RRESP/BRESP=SLVERR. For the case when the slave does not use SLVERR signal, an OKAY response is sent back to master by default.

3.11. AMBA 4 AXI Memory-Mapped Interface Support (version 2.0)

Platform Designer allows memory-mapped connections between AMBA 4 AXI components, AMBA 4 AXI and AMBA 3 AXI components, and AMBA 4 AXI and Avalon interfaces with unique or exceptional support.

3.11.1. Burst Support

Platform Designer supports INCR bursts up to 256 beats. Platform Designer converts long bursts to multiple bursts in a packet with each burst having a length less than or equal to MAX_BURST when going to AMBA 3 AXI 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 AMBA 3 AXI slaves as destinations are shortened to multiple bursts, with each burst length less than or equal to 16. Bursts with AMBA 4 AXI slaves as destinations are not shortened.

3.11.2. QoS

Platform Designer 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 AMBA 3 AXI and Avalon masters have a default value of \(4'b0000\), which indicates that the transactions are not part of the QoS flow. QoS values are not used for slaves that do not support QoS.

For Platform Designer 14.0, there are no programmable QoS registers or compile-time QoS options for a master that overrides its real or default value.

3.11.3. Regions

For Platform Designer 14.0, there is no support for the optional regions feature. AMBA 4 AXI 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.

3.11.4. Write Response Dependency

Write response dependency as specified in the Arm AMBA Protocol Specifications for AMBA 4 AXI is not supported.

Related Information

Arm AMBA Protocol Specifications
3.11.5. AWCACHE and ARCACHE

For AMBA 4 AXI, Platform Designer meets the requirement for modifiable and non-modifiable transactions. The modifiable bit refers to ARCACHE[1] and AWCACHE[1].

3.11.6. Width Adaptation and Data Packing in Platform Designer

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 AMBA 3 AXI, AMBA 4 AXI, 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, Platform Designer 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 AMBA 3 AXI, AMBA 4 AXI, or APB masters or slaves.

3.11.7. Ordering Model

Out of order support is not implemented in Platform Designer, version 14.0. Platform Designer 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.

\[(\text{AWCACHE}[1] = 0 \text{ or } \text{ARCACHE}[1] = 0)\] 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.

3.11.8. Read and Write Allocate

Read and write allocate does not apply to Platform Designer interconnect, which does not have caching features, and always receives responses from an endpoint.

3.11.9. Locked Transactions

Locked transactions are not supported for Platform Designer, version 14.0.
3.11.10. Memory Types

For AMBA 4 AXI, Platform Designer 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 Platform Designer interconnect always identifies transactions as being non-bufferable.

3.11.11. Mismatched Attributes

There are rules for how multiple masters issue cache values to a shared memory region. The interconnect meets requirements if signals are not modified.

3.11.12. Signals

Platform Designer supports up to 64-bits for the `BUSER`, `WUSER` and `RUSER` sideband signals. AMBA 4 AXI 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**

Arm AMBA Protocol Specifications

3.12. AMBA 4 AXI Streaming Interface Support (version 1.0)

3.12.1. Connection Points

Platform Designer allows you to connect an AMBA 4 AXI-Stream interface to another AMBA 4 AXI-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.

3.12.1.1. AMBA 4 AXI Streaming Connection Point Parameters

**Table 59. AMBA 4 AXI 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>
3.12.1.2. AMBA 4 AXI Streaming Connection Point Signals

Table 60. AMBA 4 AXI-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$^{(1)}$</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$^{(2)}$</td>
<td>1:8</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tdest$^{(3)}$</td>
<td>1:4</td>
<td>Output</td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>tuser$^{(4)}$</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>

3.12.2. Adaptation

AMBA 4 AXI-Stream adaptation support is not available. AMBA 4 AXI-Stream master and slave interface signals and widths must match.

3.13. AMBA 4 AXI-Lite Protocol Specification Support (version 2.0)

AMBA 4 AXI-Lite is a sub-set of AMBA 4 AXI. It is suitable for simpler control register-style interfaces that do not require the full functionality of AMBA 4 AXI.

Platform Designer 14.0 supports the following AMBA 4 AXI-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.

3.13.1. AMBA 4 AXI-Lite Signals

Platform Designer supports all AMBA 4 AXI-Lite interface signals. All signals are required.

---

$^{(1)}$ integer in multiple of bytes

$^{(2)}$ maximum 8-bits

$^{(3)}$ maximum 4-bits

$^{(4)}$ number of bits in multiple of the number of bytes of tdata
### 3.13.2. AMBA 4 AXI-Lite Bus Width

AMBA 4 AXI-Lite masters or slaves must have either 32-bit or 64-bit bus widths. Platform Designer interconnect inserts a width adapter if a master and slave pair have different widths.

### 3.13.3. AMBA 4 AXI-Lite Outstanding Transactions

AXI-Lite supports outstanding transactions. The options to control outstanding transactions is set in the parameter editor for the selected component.

### 3.13.4. AMBA 4 AXI-Lite IDs

AMBA 4 AXI-Lite does not support IDs. Platform Designer performs ID reflection inside the slave agent.

### 3.13.5. Connections Between AMBA 3 AXI, AMBA 4 AXI and AMBA 4 AXI-Lite

#### 3.13.5.1. AMBA 4 AXI-Lite Slave Requirements

For an AMBA 4 AXI-Lite slave side, the master can be any master interface type, such as an Avalon (with bursting), AMBA 3 AXI, or AMBA 4 AXI. Platform Designer allows the following connections and inserts adapters, if needed.

- **Burst adapter**—Avalon and AMBA 3 AXI and AMBA 4 AXI bursting masters require a burst adapter to shorten the burst length to 1 before sending a transaction to an AMBA 4 AXI-Lite slave.
- Platform Designer interconnect uses a width adapter for mismatched data widths.
- Platform Designer interconnect performs ID reflection inside the slave agent.
- An AMBA 4 AXI-Lite slave must have an address width of at least 12-bits.
- AMBA 4 AXI-Lite does not have the AXSIZE parameter. Narrow master to a wide AMBA 4 AXI-Lite slave is not supported. For masters that support narrow-sized bursts, for example, AMBA 3 AXI and AMBA 4 AXI, a burst to an AMBA 4 AXI-Lite slave must have a burst size equal to or greater than the slave's burst size.

#### 3.13.5.2. AMBA 4 AXI-Lite Data Packing

Platform Designer interconnect does not support AMBA 4 AXI-Lite data packing.

---

**Table 61. AMBA 4 AXI-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>
3.13.6. AMBA 4 AXI-Lite Response Merging

When Platform Designer 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.

3.14. Port Roles (Interface Signal Types)

Each interface defines 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.

3.14.1. AXI Master Interface Signal Types

Table 62. 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>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>bready</td>
<td>output</td>
<td>1</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>

### 3.14.2. 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>
<tr>
<td>awcache</td>
<td>input</td>
<td>4</td>
</tr>
<tr>
<td>awid</td>
<td>input</td>
<td>1 - 18</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

### 3.14.3. AMBA 4 AXI Master Interface Signal Types

**Table 64.** AMBA 4 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>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>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<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>rready</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>rresp</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>
</tbody>
</table>

*continued...*
### 3.14.4. AMBA 4 AXI Slave Interface Signal Types

Table 65. AMBA 4 AXI Slave Interface Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<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>
<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>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>input</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>
</tbody>
</table>
### AMBA 4 AXI-Stream Master and Slave Interface Signal Types

#### Table 66. AMBA 4 AXI-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>tsstrb</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>
3.14.6. ACE-Lite Interface Signal Roles

Table 67.  ACE-Lite Interface Signal Roles

<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>arsnoop</td>
<td>4 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>ardomain</td>
<td>2 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>arbar</td>
<td>2 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>awsnoop</td>
<td>3 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>awdomain</td>
<td>2 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>awbar</td>
<td>2 bits</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>awunique</td>
<td>1 bit</td>
<td>Output</td>
<td>Input</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.14.7. APB Interface Signal Types

Table 68.  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>psrverr</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>paddr3l</td>
<td>1</td>
<td>output</td>
<td>input</td>
<td>no</td>
</tr>
</tbody>
</table>


Signal roles define the signal types that Avalon-MM master and slave ports allow.

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.
The following table lists signal roles for the Avalon-MM interface:

### Table 69.  
**Avalon-MM Signal Roles**

Some 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>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>1 - 64</td>
<td>Master → Slave</td>
<td>No</td>
<td>Masters: By default, the address signal represents a byte address. The value of the address must align 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. From the perspective of the slave, each slave access is for a word of data. For example, <code>address = 0</code> selects the first word of the slave. <code>address = 1</code> 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>No</td>
<td>Enables one or more specific byte lanes 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 <code>&lt;n&gt;</code> of byteenable indicates whether byte <code>&lt;n&gt;</code> is being written to. During writes, byteenable specify which bytes are being written to. Other bytes should be ignored by the slave. During reads, byteenable 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.</td>
</tr>
<tr>
<td>debugaccess</td>
<td>1</td>
<td>Master → Slave</td>
<td>No</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>No</td>
<td>Asserted to indicate a read transfer. If present, readdata is required.</td>
</tr>
<tr>
<td>read_n</td>
<td>1</td>
<td>Master → Slave</td>
<td>No</td>
<td>Asserted to indicate a read transfer. If present, readdata is required.</td>
</tr>
<tr>
<td>readdata</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
<td>Slave → Master</td>
<td>No</td>
<td>The readdata driven from the slave to the master in response to a read transfer. Required for interfaces that support reads.</td>
</tr>
</tbody>
</table>
| response [1:0] | 2            | Slave → Master | No       | The response signal is an optional signal that carries the response status.  
*Note:* Because the signal is shared, an interface cannot issue or accept a write response and a read response in the same clock cycle.  
- 00: OKAY—Successful response for a transaction.  
- 01: RESERVED—Encoding is reserved.  
- 10: SLAVEERROR—Error from an endpoint slave. Indicates an unsuccessful transaction.  
- 11: DECODEERROR—Indicates attempted access to an undefined location. |

**continued...**
For read responses:
- One response is sent with each readdata. A read burst length of N results in N responses. Fewer responses are not valid, even in the event of an error. The response signal value may 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:
- 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.
- If writeresponsevalid is present, all write commands must be completed with write responses.

### Signal Role Table

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>1</td>
<td>Master → Slave</td>
<td>No</td>
<td>Asserted to indicate a write transfer. If present, writedata is required.</td>
</tr>
<tr>
<td>writedata</td>
<td>8, 16, 32, 64, 128, 256, 512, 1024</td>
<td>Master → Slave</td>
<td>No</td>
<td>Data for write transfers. The width must be the same as the width of readdata if both are present. Required for interfaces that support writes.</td>
</tr>
<tr>
<td>lock</td>
<td>1</td>
<td>Master → Slave</td>
<td>No</td>
<td>Lock ensures that once a master wins arbitration, the winning master maintains access to the slave for multiple transactions. Lock asserts coincident with the first read or write of a locked sequence of transactions. Lock deasserts on the final transaction of a locked sequence of transactions. lock assertion does not guarantee that arbitration is won. After the lock-asserting master has been granted, that master retains grant until lock 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.</td>
</tr>
</tbody>
</table>

### Wait-State Signals

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitrequest</td>
<td>1</td>
<td>Slave → Master</td>
<td>No</td>
<td>A slave asserts waitrequest when 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 must remain constant except for beginbursttransfer. For a timing diagram illustrating the beginbursttransfer signal, refer to the figure in Read Bursts.</td>
</tr>
<tr>
<td>waitrequest_n</td>
<td>1</td>
<td>Slave → Master</td>
<td>No</td>
<td>continued...</td>
</tr>
</tbody>
</table>
### Signal Role

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitrequest</td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

### Pipeline Signals

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readdatavalid</td>
<td>1</td>
<td>Slave → Master</td>
<td>No</td>
<td>Used for variable-latency, pipelined read transfers. When asserted, indicates that the readdata signal contains valid data. For a read burst with burstcount value (&lt;n&gt;), the readdatavalid signal must be asserted (&lt;n&gt;) times, once for each readdata item. There must be at least one cycle of latency between acceptance of the read and assertion of readdatavalid. For a timing diagram illustrating the readdatavalid signal, refer to Pipelined Read Transfer with Variable Latency. A slave may assert readdatavalid to transfer data to the master independently of whether 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>1</td>
<td>Slave → Master</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeresponsevalid</td>
<td>1</td>
<td>Slave → Master</td>
<td>No</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>
</tbody>
</table>

### Burst Signals

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>burstcount</td>
<td>(1 \leq \leq 11)</td>
<td>Master → Slave</td>
<td>No</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^{(\leq-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; &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>

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginbursttransfer</td>
<td>1</td>
<td>Interconnect → Slave</td>
<td>No</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. <strong>Warning:</strong> do not use this signal. This signal exists to support legacy memory controllers.</td>
</tr>
</tbody>
</table>
### 3.14.9. 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 70. 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>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>channel</td>
<td>1 – 128</td>
<td>Source → Sink</td>
<td>No</td>
<td>The channel number for data being transferred on the current cycle. If an interface supports the channel signal, the interface must also define the maxChannel parameter.</td>
</tr>
<tr>
<td>data</td>
<td>1 – 4,096</td>
<td>Source → Sink</td>
<td>No</td>
<td>The data signal from the source to the sink, typically carries the bulk of the information being transferred. Parameters further define the contents and format of the data signal.</td>
</tr>
<tr>
<td>error</td>
<td>1 – 256</td>
<td>Source → Sink</td>
<td>No</td>
<td>A bit mask to mark errors affecting the data being transferred in the current cycle. A single bit of the error signal masks each of the errors the component recognizes. The errorDescriptor defines the error signal properties.</td>
</tr>
<tr>
<td>ready</td>
<td>1</td>
<td>Sink → Source</td>
<td>No</td>
<td>Asserts high to indicate that the sink can accept data. ready is asserted by the sink on cycle ( n ) to mark cycle ( n + \text{readyLatency} ) as a ready cycle. The source may only assert valid and transfer data during ready cycles. Sources without a ready input do not support backpressure. Sinks without a ready output never need to backpressure.</td>
</tr>
<tr>
<td>valid</td>
<td>1</td>
<td>Source → Sink</td>
<td>No</td>
<td>The source asserts this signal to qualify all other source to sink signals. The sink samples data and other source-to-sink signals on ready cycles where valid is asserted. All other cycles are ignored. Sources without a valid output implicitly provide valid data on every cycle that a sink is not asserting backpressure. Sinks without a valid input expect valid data on every cycle that they are not backpressuring.</td>
</tr>
<tr>
<td><strong>Packet Transfer Signals</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td>1 – 5</td>
<td>Source → Sink</td>
<td>No</td>
<td>Indicates the number of symbols that are empty, that is, do not represent valid data. The empty signal is not necessary on interfaces where there is one symbol per beat.</td>
</tr>
<tr>
<td>endofpacket</td>
<td>1</td>
<td>Source → Sink</td>
<td>No</td>
<td>Asserted by the source to mark the end of a packet.</td>
</tr>
<tr>
<td>startofpacket</td>
<td>1</td>
<td>Source → Sink</td>
<td>No</td>
<td>Asserted by the source to mark the beginning of a packet.</td>
</tr>
</tbody>
</table>
### 3.14.10. Avalon Clock Source Signal Roles

An Avalon Clock source interface drives a clock signal out of a component.

**Table 71. 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>

### 3.14.11. Avalon Clock Sink Signal Roles

A clock sink provides a timing reference for other interfaces and internal logic.

**Table 72. 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>

### 3.14.12. Avalon Conduit Signal Roles

**Table 73. 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 bidirectional</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 Platform Designer system provided the roles and widths match and the directions are opposite.</td>
</tr>
</tbody>
</table>

### 3.14.13. 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 74. 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. When request is asserted and grant is deasserted, request is requesting access for the current cycle. When request is asserted and grant is asserted, request is requesting access for the next cycle. Consequently, request should be deasserted on the final cycle of an access. The request signal deasserts in the last cycle of a bus access. The request signal can reassert 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>
</tbody>
</table>

continued...
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.

<table>
<thead>
<tr>
<th>Signal Role</th>
<th>Width</th>
<th>Direction</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grant</td>
<td>1</td>
<td>Slave → Master</td>
<td>Yes</td>
<td>When asserted, indicates that a tristate conduit master has access to perform transactions. The grant signal asserts in response to the request signal. The grant signal 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>&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>

**Table 75. 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>read_n</td>
<td></td>
<td></td>
<td></td>
<td>present, data must also be used.</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</td>
</tr>
<tr>
<td>write_n</td>
<td></td>
<td></td>
<td></td>
<td>from a master. If present, data must also be present,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and writebyteenable cannot be present.</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</td>
</tr>
<tr>
<td>chipselect_n</td>
<td></td>
<td></td>
<td></td>
<td>chipselect is asserted. chipselect is always present in combination with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>read or write</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</td>
</tr>
<tr>
<td>outputenable_n</td>
<td></td>
<td></td>
<td></td>
<td>drive its data lines otherwise data contention may occur.</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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>During read transfers the slave device drives the data lines, and the FPGA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>captures the data signals and provides them to the master.</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 lanes during transfers. Each bit in byteenable</td>
</tr>
<tr>
<td>byteenable_n</td>
<td></td>
<td></td>
<td></td>
<td>corresponds to a byte lane in data. During writes, byteenables specify</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>which bytes the master is writing to the slave. During reads, byteenables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>indicates which bytes the master is reading. Slaves that simply return</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>data with no side effects are free to ignore byteenables during reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When more than one byte lane is asserted, all asserted lanes are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>guaranteed to be adjacent. The number of adjacent lines must be a power of 2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and the specified bytes must be aligned on an address boundary for the size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of the data. The following are legal values for a 32-bit slave:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111 writes full 32 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011 writes lower 2 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1100 writes upper 2 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001 writes byte 0 only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010 writes byte 1 only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0100 writes byte 2 only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000 writes byte 3 only</td>
</tr>
</tbody>
</table>

*continued...*
### 3.14.15. Avalon Interrupt Sender Signal Roles

**Table 76. 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>1-32</td>
<td>Output</td>
<td>Yes</td>
<td>Interrupt Request. An interrupt sender drives an interrupt signal to an interrupt receiver.</td>
</tr>
<tr>
<td>irq_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.14.16. Avalon Interrupt Receiver Signal Roles

**Table 77. 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>

### 3.15. Platform Designer Interconnect Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
|                  |                             | • Updated location of Limit interconnect pipeline stages to option in Platform Designer GUI.  
|                  |                             | • In Avalon Memory-Mapped Interface Signal Roles, added consecutive byte-enable support.  
|                  |                             | • Specified minimum duration of reset that the Platform Design Interconnect requires to work correctly. |
| 2018.06.15       | 18.0.0                      | Clarified behavior of Error Correction Coding (ECC) in Interconnect. |
| 2017.11.06       | 17.1.0                      | • Changed instances of Qsys to Platform Designer (Standard)  
|                  |                             | • Updated information about the Reset Sequencer. |
| 2015.11.02       | 15.1.0                      | Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04       | 15.0.0                      | • Fixed Priority Arbitration.  
|                  |                             | • Added topic: Read and Write Responses.  
|                  |                             | • Added topic: Error Correction Coding (ECC) in Qsys Interconnect.  
|                  |                             | • Added: response [1:0], Avalon Memory-Mapped Interface Signal Roles.  
|                  |                             | • Added writeresponsevalid, Avalon Memory-Mapped Interface Signal Roles. |

---

Note: All Avalon signals are active high. Avalon signals that can also be asserted low list both versions in the **Signal Role** column.
### Related Information

**Documentation Archive**

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.
4. Platform Designer System Design Components

You can use Platform Designer IP components to create Platform Designer systems. Platform Designer interfaces include components appropriate for streaming high-speed data, reading and writing registers and memory, controlling off-chip devices, and transporting data between components.

*Note:* Intel now refers to Qsys as Platform Designer (Standard).

Platform Designer supports Avalon, AMBA 3 AXI (version 1.0), AMBA 4 AXI (version 2.0), AMBA 4 AXI-Lite (version 2.0), AMBA 4 AXI-Stream (version 1.0), and AMBA 3 APB (version 1.0) interface specifications.

**Related Information**
- Creating a System with Platform Designer on page 10
- Platform Designer Interconnect on page 124
- AMBA Protocol Specifications
- Embedded Peripherals IP User Guide
- Avalon Interface Specifications

4.1. Bridges

Bridges affect the way Platform Designer transports data between components. You can insert bridges between master and slave interfaces to control the topology of a Platform Designer system, which affects the interconnect that Platform Designer 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 Platform Designer, one or more master interfaces from other components connect to the bridge slave. The bridge master connects to one or more slave interfaces on other components.
Figure 108. Using a Bridge in a Platform Designer System

In this example, three masters have logical connections to three slaves, although physically each master connects only to the bridge. Transfers initiated to the slave propagate to the master in the same order in which the transfers are initiated on the slave.

4.1.1. 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 Platform Designer 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.
4.1.2. 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 Platform Designer system, the DC FIFO is automatically inserted in the Platform Designer 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. Do 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.
4.1.2.1. 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 Platform Designer interconnect and allow the Intel Quartus Prime Fitter to optimize paths that require minimal propagation delay.

**Figure 110. Avalon-MM Clock Crossing Bridge**
### 4.1.2.2. Avalon-MM Clock Crossing Bridge Parameters

#### Table 78. 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 Data width 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 timing analysis.</td>
</tr>
<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 timing analysis.</td>
</tr>
</tbody>
</table>

### 4.1.3. 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 is 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.
4. Platform Designer System Design Components

Figure 111. 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.

4.1.4. 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.

### 4.1.4.1. 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 those words must be requested 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.

4.1.4.2. Avalon-MM Unaligned Burst Expansion Bridge Parameters

Figure 113. Avalon-MM Unaligned Burst Expansion Bridge Parameter Editor

Table 79. 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>
<tr>
<td>Maximum pending read transactions</td>
<td>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 is between 4 and 32. The limit for maximum queued transactions is 64.</td>
</tr>
<tr>
<td>Width of slave to optimize for</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>
</tbody>
</table>

continued...
### 4.1.4.3. Avalon-MM Unaligned Burst Expansion Bridge Example

**Figure 114. 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.

### 4.1.5. Bridges Between Avalon and AXI Interfaces

When designing a Platform Designer 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 115. 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.

**4.1.6. AXI Bridge**

With an AXI bridge, you can influence the placement of resource-intensive components, such as the width and burst adapters. Depending on its use, an AXI bridge may reduce throughput and concurrency, in return for higher $f_{MAX}$ and less logic.

You can use an AXI bridge to group different parts of your Platform Designer system. Other parts of the system can then 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 Platform Designer systems.

**Example 9. Reducing the Number of Adapters by Adding a Bridge**

The figure shows a system with a single AXI master and three AXI slaves. It also has various interconnect components, such as routers, demultiplexers, and multiplexers. Two of the slaves have a narrower data width than the master; 16-bit slaves versus a 32-bit master.
In this system, Platform Designer interconnect creates four width adapters and four
burst adapters to access the two slaves.

You can improve resource usage by adding an AXI bridge. Then, Platform Designer
needs to add only two width adapters and two burst adapters; one pair for the read
channels, and another pair for the write channel.
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.

Interconnect_0

Interconnect_1

The figure shows the same system with an AXI bridge component, and the decrease in the number of width and burst adapters. Platform Designer creates only two width adapters and two burst adapters, as compared to the four width adapters and four burst adapters in the previous figure.

Even though this system includes more components, the overall system performance improves because there are fewer resource-intensive width and burst adapters.

4.1.6.1. AXI Bridge Signal Types

Based on parameter selections that you make for the AXI Bridge component, Platform Designer instantiates either the AMBA 3 AXI or AMBA 3 AXI master and slave interfaces into the component.
Note: In AMBA 3 AXI, aw/aruser accommodates sideband signal usage by hard processor systems (HPS).

Table 80. Sets of Signals for the AXI Bridge Based on the Protocol

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>AMBA 3 AXI</th>
<th>AMBA 3 AXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>awid / arid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awaddr / araddr</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awlen / arlen</td>
<td>yes (4-bit)</td>
<td>yes (8-bit)</td>
</tr>
<tr>
<td>awsize / arsize</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awburst / arburst</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awlock / arlock</td>
<td>yes</td>
<td>yes (1-bit optional)</td>
</tr>
<tr>
<td>awcache / arcache</td>
<td>yes (2-bit)</td>
<td>yes (optional)</td>
</tr>
<tr>
<td>awprot / arprot</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awuser / aruser</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awvalid / arvalid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awready / arready</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>awqos / arqos</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>awregion / arregion</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>wid</td>
<td>yes</td>
<td>no (optional)</td>
</tr>
<tr>
<td>wdata / rdata</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>wstrb</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>wlast / rvalid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>wvalid / rlast</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>wready / rready</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>wuser / ruser</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>bid / rid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bresp / rresp</td>
<td>yes</td>
<td>yes (optional)</td>
</tr>
<tr>
<td>bvalid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bready</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

4.1.6.2. AXI Bridge Parameters

In the parameter editor, you can customize the parameters for the AXI bridge according to the requirements of your design.
Figure 118. AXI Bridge Parameter Editor

![AXI Bridge Parameter Editor](image)

Table 81. AXI Bridge Parameters

<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>AMBA 3</td>
<td>Specifies the AXI version and signals that Platform Designer generates for the slave and master interfaces of the bridge.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AXI or AMBA 3 AXI</td>
<td></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>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>
### 4.1.6.3. AXI Bridge Slave and Master Interface Parameters

#### Table 82. 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 Platform Designer 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 Platform Designer 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 or the interconnect must apply backpressure.

### 4.1.7. AXI Timeout Bridge

The AXI Timeout Bridge allows your system to recover when it freezes, and facilitates debugging. 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 freeze. If a slave does not accept a command or respond to a command it accepted, its master can wait indefinitely.

**Figure 119. 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 freeze, place the bridge near the slave. If the master attempts to communicate with a slave that freezes, 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 place the bridge at the slave’s side and you have multiple slaves connected to the same master, you do not get the full address.
4.1.7.1. AXI Timeout Bridge Stages

A timeout occurs when the internal timer in the bridge exceeds the specified number of cycles within which a burst must complete from start to end.
Figure 121. AXI Timeout Bridge Stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No more outstanding commands</td>
</tr>
<tr>
<td>B</td>
<td>A read/write times out</td>
</tr>
<tr>
<td>C</td>
<td>The AXI Timeout Bridge is notified that the slave is reset.</td>
</tr>
<tr>
<td></td>
<td><strong>A</strong> Slave is functional - The bridge passes through all bursts.</td>
</tr>
<tr>
<td></td>
<td><strong>B</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>C</strong> Slave is reset - The bridge does not accept new commands, and responds only to commands that are outstanding.</td>
</tr>
</tbody>
</table>

- 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 is 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 83. 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) (AMBA 3 AXI-Lite), and Interrupt. Platform Designer allows the AXI Timeout Bridge to connect to any AMBA 3 AXI, AMBA 3 AXI, 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. Do not 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 84. CSR Interrupt Status Information for the AXI Timeout Bridge

<table>
<thead>
<tr>
<th>Address</th>
<th>Attribute</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>write-only</td>
<td>Slave is reset</td>
</tr>
<tr>
<td>0x4</td>
<td>read-only</td>
<td>Timed out operation</td>
</tr>
<tr>
<td>0x8 through 0xF</td>
<td>read-only</td>
<td>Timed out address</td>
</tr>
</tbody>
</table>

4.1.7.2. AXI Timeout Bridge Parameters

Table 85. 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>
4.1.8. Address Span Extender

The **Address Span Extender** allows memory-mapped master interfaces to access a larger or smaller address map than the width of their address signals allows. 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 with 1-64 bit address windows.

**Figure 122. Address Span Extender**

If a processor can address only 2 GB of an address span, and your system contains 4 GB of memory, the address span extender can provide two, 2 GB windows in the 4 GB memory address space. This issue sometimes occurs with Intel SoC devices.

For example, an HPS subsystem in an SoC device can address only 1 GB of an address span within the FPGA, using the HPS-to-FPGA bridge. The address span extender enables the SoC device to address all the address space in the FPGA using multiple 1 GB windows.

4.1.8.1. CTRL Register Layout

The control registers consist of one 64-bit register for each window, where you specify the window's base address. For example, if `CTRL_BASE` is the base address of the control register, and address span extender contains 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).
4.1.8.2. Address Span Extender Parameters

Table 86. Address Span Extender Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath Width</td>
<td>Width of write data and read data signals.</td>
</tr>
<tr>
<td>Expanded Master Byte Address Width</td>
<td>Width of the master byte address port. That is, the address span size of all the downstream slaves that attach to the address span extender.</td>
</tr>
<tr>
<td>Slave Word Address Width</td>
<td>Width of the slave word address port. That is, the address span size of the downstream slaves that the upstream master accesses.</td>
</tr>
<tr>
<td>Burstcount Width</td>
<td>Burst count port width of the downstream slave and the upstream master that attach to the address span extender.</td>
</tr>
<tr>
<td>Number of sub-windows</td>
<td>The slave port can represent one or more windows in the master address span. You can subdivide the slave address span into ( N ) equal spans in ( N ) sub-windows. A remapping register in the CSR slave represents each sub-window, and configure the base address that each sub-window remaps to. The address span extender replaces the upper bits of the address with the stored index value in the remapping register before the master initiates a transaction.</td>
</tr>
<tr>
<td>Enable Slave Control Port</td>
<td>Dictates run-time control over the sub-window indexes. If you can define static re-mappings that do not need any change, you do not need to enable this CSR slave.</td>
</tr>
<tr>
<td>Maximum Pending Reads</td>
<td>Sets the bridge slave's <code>maximumPendingReadTransactions</code> property. In certain system configurations, you must increase this value to improve performance. This value usually aligns with the properties of the downstream slaves that you attach to this bridge.</td>
</tr>
</tbody>
</table>

4.1.8.3. Calculating the Address Span Extender Slave Address

The diagram describes how Platform Designer calculates the slave address. In this example, the address span extender is configured with a 28-bit address space for slaves. The upper 2 bits \([27:26]\) are used to select the control registers.

The lower 26 bits \([25:0]\) originate from the address span extender’s data port, and are the offset into a particular window.

Figure 123. Address Span Extender

![Address Span Extender Diagram](image-url)
4.1.8.4. Using the Address Span Extender

This example shows when and how to use address span extender component in your Platform Designer design.

**Figure 124. Block Diagram with Address Span Extender**

In the above design, a 32-bit master shares 4 GB SDRAM with an external streaming interface. The master has the path to access streaming data from the SDRAM DDR memory. However, if you connect the whole 32-bit address bus of the master to the SDRAM DDR memory, you cannot connect the master to peripherals such as LED or UART. To avoid this situation, you can implement the address span extender between the master and DDR memory. The address span extender allows the master to access the SDRAM DDR memory and the peripherals at the same time.

To implement address span extender for the above example, you can divide the address window of the address span extender into two sub-windows of 512 MB each. The sub-window 0 is for the master program area. You can dynamically map the sub-window 1 to any area other than the program area.

You can change the offset of the address window by setting the base address of sub-window 1 to the control register of the address span extender. However, you must make sure that the sub-window address span masks the base address. You can choose any arbitrary base address. If you set the value 0xa000_0000 to the control register, Platform Designer maps the sub-window 1 to 0xa000_0000.
Table 87. CSR Mapping Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000_0000</td>
<td>0x0000_0000</td>
</tr>
<tr>
<td>0x8000_0008</td>
<td>0xa000_0000</td>
</tr>
</tbody>
</table>

Figure 125. Memory mapping for Address Span Extender

The table below indicates the Platform Designer parameter settings for this address span extender example.

Table 88. Parameter Settings for the Address Span Extender Example

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath Width</td>
<td>32 bits</td>
<td>The CPU has 32-bits data width and the SDRAM DDR memory has 512-bits data width. Since the transaction between the master and SDRAM DDR memory is minimal, set the datapath width to align with the upstream master.</td>
</tr>
<tr>
<td>Expanded Master Byte Address</td>
<td>32 bits</td>
<td>The address span extender has a 4 GB address span.</td>
</tr>
<tr>
<td>Slave Word Address Width</td>
<td>18 bits</td>
<td>There are two 512 MB sub-windows in reserve for the master. The number of bytes over the data word width in the Datapath Properties (4 bytes for this example) accounts for the slave address.</td>
</tr>
<tr>
<td>Burstcount Width</td>
<td>4 bits</td>
<td>The address span extender must handle up to 8 words burst in this example.</td>
</tr>
<tr>
<td>Number of sub-windows</td>
<td>2</td>
<td>Address window of the address span extender has two sub-windows of 512 MB each.</td>
</tr>
<tr>
<td>Enable Slave Control Port</td>
<td>true</td>
<td>The address span extender component must have control to change the base address of the sub-window.</td>
</tr>
<tr>
<td>Maximum Pending Reads</td>
<td>4</td>
<td>This number is the same as SDRAM DDR memory burst count.</td>
</tr>
</tbody>
</table>
Note: You can view the address span extender connections in the System Contents tab. The windowed slave port and control port connect to the master. The expanded master port connects to the SDRAM DDR memory.

4.1.8.5. 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. Platform Designer structures the logic so that Platform Designer can optimize and remove bits that are not needed.

If Burstcount Width is greater than 1, Platform Designer processes the read burst in a single cycle, and assumes all byteenable signals are asserted on every cycle.

4.1.8.6. 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 the Nios II processor 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 the Nios II processor. This technique allows the Nios II processor to communicate with HPS peripherals.
In the example, a Nios II processor has an address span extender from address 0x400000 to 0x800000. There is a window within the address span extender starting at 0x100000. Within the address span extender’s address space there is a slave at base address 0x1100000. The slave appears to the Nios II processor as being at address:

\[
0x110000 - 0x100000 + 0x40000 = 0x050000
\]

Figure 127. Nios II Support and the Address Span Extender

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.

4.2. Error Response Slave

The Error Response Slave provides a predictable error response service for master interfaces that attempt to access an undefined memory region.

The Error Response Slave is an AMBA 3 AXI component, and appears in the Platform Designer IP Catalog under Platform Designer Interconnect.

To comply with the AXI protocol, the interconnect logic must return the DECERR error response in cases where the interconnect cannot decode slave access. Therefore, an AXI system with address space not fully decoded to slave interfaces requires the Error Response Slave.

The Error Response Slave behaves like any other component in the system, and connects to other components via translation and adaptation interconnect logic. Connecting an Error Response Slave to masters of different data widths, including Avalon or AXI-Lite masters, can increase resource usage.

An Error Response Slave can connect to clock, reset, and IRQ signals as well as AMBA 3 AXI and AMBA 4 AXI master interfaces without instantiating a bridge. When you connect an Error Response Slave to a master, the Error Response Slave accepts cycles sent from the master, and returns the DECERR error response. On the AXI interface, the Error Response Slave supports only a read and write acceptance of capability 1,
and does not support write data interleaving. The Error Response Slave can return responses when simultaneously targeted by a read and write cycle, because its read and write channels are independent.

An optional Avalon interface on the Error Response Slave provides information in a set of CSR registers. CSR registers log the required information when returning an error response.

- To set the Error Response Slave as the default slave for a master interface in your system, connect the slave to the master in your Platform Designer system.
- A system can contain more than one Error Response Slave.
- As a best practice, instantiate separate Error Response Slave components for each AXI master in your system.

Related Information

- AMBA 3 AXI Protocol Specification Support (version 1.0) on page 184
- Designating a Default Slave on page 238

4.2.1. Error Response Slave Parameters

Figure 128. Error Response Slave Parameter Editor

If you turn on Enable CSR Support (for error logging) more parameters become available.
Figure 129. Error Response Slave Parameter Editor with Enabled CSR Support

Table 89. Error Response 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>Specifies the master ID width for error logging.</td>
</tr>
<tr>
<td>AXI address width</td>
<td>8-64 bits</td>
<td>Specifies 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></td>
<td></td>
<td>address width required in the system.</td>
</tr>
<tr>
<td>AXI data width</td>
<td>32, 64, or 128 bits</td>
<td>Specifies the data width for error logging.</td>
</tr>
<tr>
<td>Enable CSR Support (for logging)</td>
<td>On / Off</td>
<td>When turned on, instantiates an Avalon CSR interface for error logging.</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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSR logs for cycles with errors.</td>
</tr>
<tr>
<td>Register Avalon CSR inputs</td>
<td>On / Off</td>
<td>When turned on, controls debug access to the CSR interface.</td>
</tr>
</tbody>
</table>

4.2.2. Error Response Slave CSR Registers

The Error Response Slave with enabled CSR support provides a service to handle access violations. This service uses CSR registers for status and logging purposes.

The sequence of actions in the access violation service is equivalent for read and write access violations, but the CSR status bits and log registers are different.

4.2.2.1. Error Response Slave Access Violation Service

When an access violation occurs, and the CSR port is enabled:

1. The Error Response Slave generates an interrupt:
   - For a read access violation, the Error Response Slave sets the Read Access Violation Interrupt register bit in the Interrupt Status register.
— For a write access violation, the Error Response Slave sets the Write Access Violation Interrupt register bit in the Interrupt Status register.

2. The Error Response Slave transfers transaction information to the access violation log FIFO. The amount of information that the FIFO can handle is given by the Error Log Depth parameter.

You define the Error Log Depth in the Parameter Editor, when you enable CSR Support.

3. Software reads entries of the access violation log FIFO until the corresponding cycle log valid bit is cleared, and then exits the service routine.

— The Read cycle log valid bit is in the Read Access Violation Log CSR Registers.

— The Write cycle log valid bit is in the Write Access Violation Log CSR Registers.

4. The Error Response Slave clears the interrupt bit when there are no access violations to report.

Some special cases are:

• If any error occurs when the FIFO is full, the Error Response Slave sets the corresponding Access Violation Interrupt Overflow register bit (bits 2 and 3 of the Status Register for write and read access violations, respectively). Setting this bit means that not all error entries were written to the access violation log.

• After Software reads an entry in the Access Violation log, the Error Response Slave can write a new entry to the log.

• Software can specify the number of entries to read before determining that the access violation service is taking too long to complete, and exit the routine.

### 4.2.2.2. CSR Interrupt Status Registers

#### Table 90. CSR Interrupt Status Registers

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bits</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31:4</td>
<td></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>3</td>
<td>RWIC</td>
<td>0</td>
<td>Read Access Violation Interrupt Overflow register Asserted when a read access causes the Interconnect to return a DECERR response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RWIC</td>
<td>0</td>
<td>Write Access Violation Interrupt Overflow register Asserted when a write access causes the Interconnect to return a DECERR response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>RWIC</td>
<td>0</td>
<td>Read Access Violation Interrupt register Asserted when a read access causes the Interconnect to return a DECERR response. Cleared by setting the bit to 1.</td>
<td></td>
</tr>
</tbody>
</table>

continued...
### 4.2.2.3. CSR Read Access Violation Log Registers

The CSR read access violation log settings are valid only when an associated read interrupt register is set. Read this set of registers until the validity bit is cleared.

**Table 91. CSR Read Access Violation Log Registers**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bits</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>31:13</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>12:11</td>
<td>RO</td>
<td>0</td>
<td>Offending Read cycle burst type: Specifies the burst type of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>10:7</td>
<td>RO</td>
<td>0</td>
<td>Offending Read cycle burst length: Specifies the burst length of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>6:4</td>
<td>RO</td>
<td>0</td>
<td>Offending Read cycle burst size: Specifies the burst size of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>3:1</td>
<td>RO</td>
<td>0</td>
<td>Offending Read cycle PROT: Specifies the PROT of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>RO</td>
<td>0</td>
<td>Read cycle log valid: Specifies the validity of the read access violation log. This bit is cleared when the interrupt register is cleared.</td>
</tr>
<tr>
<td>0x104</td>
<td>31:0</td>
<td>RO</td>
<td>0</td>
<td>Offending read cycle ID: Master ID for the cycle that causes the access violation.</td>
</tr>
<tr>
<td>0x108</td>
<td>31:0</td>
<td>RO</td>
<td>0</td>
<td>Offending read cycle target address: Target address for the cycle that causes the access violation (lower 32-bit).</td>
</tr>
<tr>
<td>0x10C</td>
<td>31:0</td>
<td>RO</td>
<td>0</td>
<td>Offending read cycle target address: Target address for the cycle that causes the access violation (upper 32-bit). Valid only if widest address in system is larger than 32 bits. Note: When this register is read, the current read access violation log is recovered from FIFO.</td>
</tr>
</tbody>
</table>

### 4.2.2.4. CSR Write Access Violation Log Registers

The CSR write access violation log settings are valid only when an associated write interrupt register is set. Read this set of registers until the validity bit is cleared.

**Table 92. CSR Write Access Violation Log**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bits</th>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x190</td>
<td>31:13</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>12:11</td>
<td>RO</td>
<td>0</td>
<td>Offending write cycle burst type: Specifies the burst type of the initiating cycle that causes the access violation.</td>
</tr>
<tr>
<td></td>
<td>10:7</td>
<td>RO</td>
<td>0</td>
<td>Offending write cycle burst length: Specifies the burst length of the initiating cycle that causes the access violation.</td>
</tr>
</tbody>
</table>
### 4.2.3. Designating a Default Slave

You can designate any slave in your Platform Designer system as the error response default slave. The default slave you designate provides an error response service for masters that attempt access to an undefined memory region.

1. In your Platform Designer 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.

#### Related Information

Specifying a Default Slave on page 48
4.3. Tri-State Components

The tri-state interface type allows you to design Platform Designer 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.

Example 10. Tri-State Conduit System to Control Off-Chip SRAM and Flash Devices

In this example, there are two generic Tri-State Conduit Controllers. The first is customized to control a flash memory. The second is customized to control an off-chip SSRAM. The Tri-State Conduit Pin Sharer multiplexes between these two controllers, and the Tri-State Conduit Bridge converts between an on-chip encoding of tri-state signals and true bidirectional signals. By default, the Tri-State Conduit Pin Sharer and Tri-State Conduit Bridge present byte addresses. Typically, each address location contains more than one byte of data.

Figure 130. Tri-State Conduit System to Control Off-Chip SRAM and Flash Devices

Address Connections from Platform Designer System to PCB

The flash device operates on 16-bit words and must ignore the least-significant bit of the Avalon-MM address. The figure 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 MB to 8 MB-1. The SSRAM responds to address range 8 MB to 10 MB-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 MB flash device accesses 16-bit words; consequently, the schematic does not connect addr[0]. The chipselect signals select between the two devices.
Figure 131. Address Connections from Platform Designer System to PCB

Note: If you create a custom tri-state conduit master with word aligned addresses, the Tri-state Conduit Pin Sharer does not change or align the address signals.

Figure 132. Tri-State Conduit System in Platform Designer

Related Information
- Avalon Tri-State Conduit Components User Guide
- Avalon Interface Specifications
4.3.1. Generic Tri-State Controller

The Generic Tri-State Controller provides a template for a controller. You can customize the tri-state controller with various parameters to reflect the behavior of an off-chip device. The following types of parameters are available for the tri-state controller:

- Width of the address and data signals
- Read and write wait times
- Bus-turnaround time
- Data hold time

**Note:** In calculating delays, the Generic Tri-State Controller chooses the larger of the bus-turnaround time and read latency. Turnaround time is measured from the time that a command is accepted, not from the time that the previous read returned data.

The Generic Tri-State Controller includes the following interfaces:

- **Memory-mapped slave interface**—This interface connects to a memory-mapped master, such as a processor.
- **Tristate Conduit Master interface**—The 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.

4.3.2. 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 133. 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 connected to a pin sharer must be in the same clock domain.

Related Information
Avalon-ST Round Robin Scheduler on page 266

4.3.3. 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.

4.4. Test Pattern Generator and Checker Cores

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 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 **Throttle Seed** is the starting value for the throttle control random number generator. Intel recommends a unique value for each instance of the test pattern generator and checker cores in a system.

### 4.4.1. Test Pattern Generator

**Figure 134. Test Pattern Generator Core**

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface, such as the number of error bits and data signal width, thus allowing you to test components with different interfaces.

The data pattern is calculated as: Symbol Value = Symbol Position in Packet XOR Data Error Mask. Data that is not organized in packets is a single stream with no beginning or end. The test pattern generator has a throttle register that is set via the Avalon-MM control interface. The test pattern generator uses the value of the throttle register in conjunction with a pseudo-random number generator to throttle the data generation rate.

**4.4.1.1. 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 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.

### 4.4.1.2. 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 data packets are supported.

### 4.4.1.3. 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 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.

### 4.4.1.4. Test Pattern Generator Functional Parameter

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

Figure 135. Test Pattern Checker

The test pattern checker core accepts data via an Avalon-ST interface and verifies it against the same predetermined pattern that the test pattern generator uses to produce the data. The test pattern checker core reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker’s Avalon-ST interface such as the number of error bits and the data signal width. This enables the ability to test components with different interfaces. The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.

The test pattern checker detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP), and signaled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

4.4.2.1. 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.

4.4.2.2. 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.
4.4.2.3. Test Pattern Checker Functional Parameter

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

4.4.2.4. 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 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.

4.4.3. 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.

4.4.3.1. HAL System Library Support

For Nios II processor users, Intel provides HAL system library drivers that allow you to initialize and access the test pattern generator and checker cores. Intel recommends you 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.

4.4.3.2. 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.

### 4.4.3.3. Register Maps for the Test Pattern Generator and Test Pattern Checker Cores

#### 4.4.3.3.1. Test Pattern Generator Control and Status Registers

**Table 93. 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 94. Test Pattern Generator Status Register Bits**

<table>
<thead>
<tr>
<th>Bits</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 95. Test Pattern Generator Control Register Bits**

<table>
<thead>
<tr>
<th>Bits</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>
<tr>
<td>[16:8]</td>
<td>THROTTLE</td>
<td>RW</td>
<td>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.</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></td>
<td></td>
</tr>
</tbody>
</table>
## Test Pattern Generator Fill Register Bits

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>BUSY</td>
<td>RO</td>
<td>A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.</td>
</tr>
<tr>
<td>[6:1]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[15:7]</td>
<td>FILL</td>
<td>RO</td>
<td>The number of commands currently in the command FIFO.</td>
</tr>
<tr>
<td>[31:16]</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Test Pattern Generator Command Registers

#### Table 97. Test Pattern Generator Command Register Map

Shows the offset for the command 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>cmd_lo</td>
</tr>
<tr>
<td>base + 1</td>
<td>cmd_hi</td>
</tr>
</tbody>
</table>

The.cmd_lo is pushed into the FIFO only when the cmd_lo register is addressed.

#### Table 98. cmd_lo Register Bits

<table>
<thead>
<tr>
<th>Bits</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 99. cmd_hi Register Bits

<table>
<thead>
<tr>
<th>Bits</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 signaled 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>
4.4.3.3. Test Pattern Checker Control and Status Registers

Table 100. 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>base + 2</td>
<td>Reserved</td>
</tr>
<tr>
<td>base + 3</td>
<td></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 101. 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 102. 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></td>
<td></td>
<td>Reserved</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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Platform Designer uses this value in conjunction with a pseudo-random number generator to throttle the data generation rate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>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></td>
<td></td>
</tr>
</tbody>
</table>

If there is no exception, reading the exception_descriptor register bit register returns 0.

Table 103. 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>
</tbody>
</table>

continued...


<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 104. **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 105. **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>

**4.4.4. 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 251
- data_source_init() on page 251
- data_source_get_id() on page 251
- data_source_get_supports_packets() on page 252
- data_source_get_num_channels() on page 252
- data_source_get_symbols_per_cycle() on page 252
- data_source_get_enable() on page 252
- data_source_set_enable() on page 253
- data_source_get_throttle() on page 253
- data_source_set_throttle() on page 253
4.4.4.1. data_source_reset()

Table 106. data_source_reset()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_source_reset(alt_u32 base);</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.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</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>

4.4.4.2. data_source_init()

Table 107. data_source_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, alt_u32 command_base);</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. command_base—Base address of the command slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Initialization is successful. 0—Initialization is unsuccessful.</td>
</tr>
<tr>
<td>Description</td>
<td>Performs the following operations to initialize the test pattern generator core:  • Resets and disables the test pattern generator core.  • Sets the maximum throttle.  • Clears all inserted errors.</td>
</tr>
</tbody>
</table>

4.4.4.3. data_source_get_id()

Table 108. data_source_get_id()

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>int data_source_get_id(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>Test pattern generator core identifier.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the test pattern generator core's identifier.</td>
</tr>
</tbody>
</table>
4.4.4.4. data_source_get_supports_packets()

Table 109. 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.</td>
</tr>
<tr>
<td></td>
<td>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>

4.4.4.5. data_source_get_num_channels()

Table 110. 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>

4.4.4.6. data_source_get_symbols_per_cycle()

Table 111. 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>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 symbols transferred in a beat.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of symbols transferred by the test pattern generator core</td>
</tr>
<tr>
<td></td>
<td>in each beat.</td>
</tr>
</tbody>
</table>

4.4.4.7. data_source_get_enable()

Table 112. data_source_get_enable()

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

...continued...
4.4.4.8. **data_source_set_enable()**

Table 113.  **data_source_set_enable()**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>void data_source_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_source_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave. value—ENABLE bit set to the value of this parameter.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>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</td>
</tr>
</tbody>
</table>

4.4.4.9. **data_source_get_throttle()**

Table 114.  **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>

4.4.4.10. **data_source_set_throttle()**

Table 115.  **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.</td>
</tr>
</tbody>
</table>

continued...
### 4.4.4.11. data_source_is_busy()

**Table 116. 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.</td>
</tr>
<tr>
<td></td>
<td>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>

### 4.4.4.12. data_source_fill_level()

**Table 117. 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>

### 4.4.4.13. data_source_send_data()

**Table 118. 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>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>cmd_base—Base address of the command slave.</td>
</tr>
<tr>
<td></td>
<td>channel—Channel to send the data.</td>
</tr>
<tr>
<td>Information Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>size—Data size.</td>
<td>flags — Specifies whether to send or suppress SOP and EOP signals. Valid values are <code>DATA_SOURCE_SEND_SOP</code>, <code>DATA_SOURCE_SEND_EOP</code>, <code>DATA_SOURCE_SEND_SUPRESS_SOP</code> and <code>DATA_SOURCE_SEND_SUPRESS_EOP</code>.</td>
</tr>
<tr>
<td>error—Value asserted on the error signal on the output interface.</td>
<td>data_error_mask—Parameter and the data are XORed together to produce erroneous data.</td>
</tr>
</tbody>
</table>

**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.

### 4.4.5. 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 256
- `data_sink_init()` on page 256
- `data_sink_get_id()` on page 256
- `data_sink_get_supports_packets()` on page 257
- `data_sink_get_num_channels()` on page 257
- `data_sink_get_symbols_per_cycle()` on page 257
- `data_sink_get_enable()` on page 257
- `data_sink_set_enable()` on page 258
- `data_sink_get_throttle()` on page 258
- `data_sink_set_throttle()` on page 258
- `data_sink_get_packet_count()` on page 259
- `data_sink_get_error_count()` on page 259
- `data_sink_get_symbol_count()` on page 259
- `data_sink_get_exception()` on page 260
- `data_sink_exception_is_exception()` on page 260
- `data_sink_exception_has_data_error()` on page 260
- `data_sink_exception_has_missing_sop()` on page 261
- `data_sink_exception_has_missing_eop()` on page 261
- `data_sink_exception_signalled_error()` on page 261
- `data_sink_exception_channel()` on page 262
### 4.4.5.1. data_sink_reset()

Table 119. 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>

### 4.4.5.2. data_sink_init()

Table 120. 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>
<tr>
<td>Returns</td>
<td>1—Initialization is successful. 0—Initialization is unsuccessful.</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 4.4.5.3. data_sink_get_id()

Table 121. 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>
4.4.5.4. data_sink_get_supports_packets()

Table 122.  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>

4.4.5.5. data_sink_get_num_channels()

Table 123.  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>

4.4.5.6. data_sink_get_symbols_per_cycle()

Table 124.  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>

4.4.5.7. data_sink_get_enable()

Table 125.  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>

continued...
### 4.4.5.8. data_sink_set_enable()

Table 126.  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>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. value—ENABLE bit is set to the value of the parameter.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>Enables the test pattern checker core.</td>
</tr>
</tbody>
</table>

### 4.4.5.9. data_sink_get_throttle()

Table 127.  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>

### 4.4.5.10. data_sink_set_throttle()

Table 128.  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>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave.</td>
</tr>
</tbody>
</table>

continued...
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.

### 4.4.5.11. data_sink_get_packet_count()

**Table 129. 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>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave. channel—Channel number.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of data packets received on the channel.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of data packets received on a channel.</td>
</tr>
</tbody>
</table>

### 4.4.5.12. data_sink_get_error_count()

**Table 130. 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>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>base—Base address of the control and status slave. channel—Channel number.</td>
</tr>
<tr>
<td>Returns</td>
<td>Number of errors received on the channel.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the number of errors received on a channel.</td>
</tr>
</tbody>
</table>

### 4.4.5.13. data_sink_get_symbol_count()

**Table 131. 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>&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>Information Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>channel</td>
<td>Channel number.</td>
</tr>
</tbody>
</table>

**Returns**

Number of symbols received on the channel.

**Description**

Retrieves the number of symbols received on a channel.

### 4.4.5.14. `data_sink_get_exception()`

**Table 132. `data_sink_get_exception()`**

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td><code>int data_sink_get_exception(alt_u32 base);</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>base</code>—Base address of the control and status slave.</td>
</tr>
<tr>
<td>Returns</td>
<td>First exception descriptor in the exception FIFO. 0—No exception descriptor found in the exception FIFO.</td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.</td>
</tr>
</tbody>
</table>

### 4.4.5.15. `data_sink_exception_is_exception()`

**Table 133. `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>
<tr>
<td>Returns</td>
<td>1—Indicates an exception. 0—No exception.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if an exception descriptor describes a valid exception.</td>
</tr>
</tbody>
</table>

### 4.4.5.16. `data_sink_exception_has_data_error()`

**Table 134. `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>
<tr>
<td>Returns</td>
<td>1—Data has errors. 0—No errors.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if an exception indicates erroneous data.</td>
</tr>
</tbody>
</table>
4.4.5.17. data_sink_exception_has_missing_sop()

Table 135.  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>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Missing SOP. 0—Other exception types.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if an exception descriptor indicates missing SOP.</td>
</tr>
</tbody>
</table>

4.4.5.18. data_sink_exception_has_missing_eop()

Table 136.  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>&lt;data_sink_util.h&gt;</td>
</tr>
<tr>
<td>Parameters</td>
<td>exception—Exception descriptor.</td>
</tr>
<tr>
<td>Returns</td>
<td>1—Missing EOP. 0—Other exception types.</td>
</tr>
<tr>
<td>Description</td>
<td>Checks if an exception descriptor indicates missing EOP.</td>
</tr>
</tbody>
</table>

4.4.5.19. data_sink_exception_signalled_error()

Table 137.  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 signaled error from the exception.</td>
</tr>
</tbody>
</table>
4.4.5.20. data_sink_exception_channel()

Table 138. 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>

4.5. Avalon-ST Splitter Core

Figure 136. Avalon-ST Splitter Core

The Avalon-ST Splitter Core allows you to replicate transactions from an Avalon-ST sink interface to multiple Avalon-ST source 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.

4.5.1. 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** option is enabled, 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** option is disabled, 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.

### 4.5.2. 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 139. 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>

### 4.5.3. Splitter Core Parameters

**Table 140. 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. Platform Designer supports 1 for some systems where no duplicated output is required.</td>
</tr>
<tr>
<td>Qualify Valid Out</td>
<td>Enabled, Disabled</td>
<td>Enabled</td>
<td>If enabled, the <code>out_valid</code> signal of all output interfaces is gated when back pressure 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>Enabled, Disabled</td>
<td>Disabled</td>
<td>Enable support of data packet transfers. Packet support includes the <code>startofpacket</code>, <code>endofpacket</code>, and <code>empty</code> signals.</td>
</tr>
<tr>
<td>Use Channel</td>
<td>Enabled, Disabled</td>
<td>Disabled</td>
<td>Enable the channel signal.</td>
</tr>
<tr>
<td>Channel Width</td>
<td>0–8</td>
<td>1</td>
<td>The width of the <code>channel</code> signal on the data interfaces. This parameter is disabled when Use Channel is set to 0.</td>
</tr>
</tbody>
</table>

continued...
### Parameter Table

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>Enabled, Disabled</td>
<td>Disabled</td>
<td>Enable 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>

#### 4.6. Avalon-ST Delay Core

**Figure 137. 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.

![Avalon-ST Delay Core Diagram](image)

The Avalon-ST 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 from 0 to 16. The change of the `in_valid` signal is reflected on the `out_valid` signal exactly $N$ cycles later.

#### 4.6.1. 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.

#### 4.6.2. Delay Core Interfaces

The Delay core supports streaming data, with optional packet, channel, and error signals. The delay core does not support backpressure.
### Table 141. 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>

### 4.6.3. Delay Core Parameters

### Table 142. 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. Platform Designer 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 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>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>
4.7. Avalon-ST Round Robin Scheduler

Figure 138. 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.

4.7.1. 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>
<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>

4.7.2. 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.

4.7.3. 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 bit of data from a channel at each transaction. To request 1 bit 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 \texttt{waitrequest} when it cannot accept new requests.

<table>
<thead>
<tr>
<th>Table 144. Avalon-ST Round Robin Scheduler Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signal</strong></td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Clock and Reset</td>
</tr>
<tr>
<td>clk</td>
</tr>
<tr>
<td>reset_n</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Avalon-MM Request Interface</td>
</tr>
<tr>
<td>request_address (( \log_2 ) Max_Channels-1:0)</td>
</tr>
<tr>
<td>request_write</td>
</tr>
<tr>
<td>request_writedata</td>
</tr>
<tr>
<td>request_waitrequest</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Avalon-ST Almost-Full Status Interface</td>
</tr>
<tr>
<td>almost_full_valid</td>
</tr>
<tr>
<td>almost_full_channel (Channel_Width-1:0)</td>
</tr>
<tr>
<td>almost_full_data (( \log_2 ) Max_Channels-1:0)</td>
</tr>
</tbody>
</table>

**4.7.4. Round Robin Scheduler Parameters**

<table>
<thead>
<tr>
<th>Table 145. Avalon-ST Round Robin Scheduler Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>Number of channels</td>
</tr>
<tr>
<td>Use almost-full status</td>
</tr>
</tbody>
</table>
4.8. Avalon Packets to Transactions Converter

Figure 139. Avalon Packets to Transactions Converter Core

The Avalon Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon-MM transactions. The core then returns Avalon-MM transaction responses to the requesting components.

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

4.8.1. Packets to Transactions Converter Interfaces

Table 146. 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>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.

4.8.2. 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.
4.8.2.1. 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>
<thead>
<tr>
<th>Table 147. Data Packet Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>[3:2]</td>
</tr>
<tr>
<td>[n:8]</td>
</tr>
</tbody>
</table>

**Response Packet Format**

| **Byte** | **Field** | **Description** |
| 0 | Transaction code | The transaction code with the most significant bit inversed. |
| 1 | Reserved | Reserved for future use. |
| [4:2] | Size | Total number of bytes read/written successfully. |

**Related Information**

Packets to Transactions Converter Interfaces on page 268

4.8.2.2. Packets to Transactions Converter Supported Transactions

The Packets to Transactions Converter core supports the following Avalon-MM transactions:

<table>
<thead>
<tr>
<th>Table 148. Packets to Transactions Converter Supported Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transaction Code</strong></td>
</tr>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0x04</td>
</tr>
<tr>
<td>0x10</td>
</tr>
<tr>
<td>0x14</td>
</tr>
<tr>
<td>0x7f</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 datapaths. 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 can 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.

### 4.8.2.3. 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.

### 4.9. 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.

After the backpressure is deasserted, the pipeline stage's source interface is deasserted and the pipeline stage asserts internally buffered data (if present). Additionally, the pipeline stage deasserts back pressure on its sink interface.
4.10. 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 datapaths 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 266
4.10.1. 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, Platform Designer cannot control or configure any aspect of the multiplexer or demultiplexer at run-time. The components cannot generate interrupts.

4.10.2. Avalon-ST Multiplexer

Figure 142. Avalon-ST Multiplexer

The Avalon-ST multiplexer takes data from a variety of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that the other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.

The multiplexer includes an optional channel signal that enables each input interface to carry channelized data. The output interface channel width is equal to:

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

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.
4.10.2.1. 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.

4.10.2.2. 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 packet transfers are supported. Packet support includes the startofpacket, endofpacket, and empty signals.
- **Channel Signal Width (bits)**—The number of bits Platform Designer 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.

4.10.2.3. 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.
4.10.3. Avalon-ST Demultiplexer

Figure 143. 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.

4.10.3.1. 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 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.
4.10.3.2. 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.

4.10.3.3. 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 your 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 144. Select Bits for the Demultiplexer**

4.11. 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.
4.11.1. Interfaces Implemented in FIFO Cores

The following interfaces are implemented in FIFO cores:

- **Avalon-ST Data Interface** on page 277
- **Avalon-MM Control and Status Register Interface** on page 277
4.11.1.1. 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 149. 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>
<tr>
<td>Channel</td>
<td>Supported, up to 255 channels.</td>
</tr>
<tr>
<td>Error</td>
<td>Configurable.</td>
</tr>
<tr>
<td>Packet</td>
<td>Configurable.</td>
</tr>
</tbody>
</table>

4.11.1.2. 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.

4.11.1.3. 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.

4.11.2. 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, the Use store and forward parameter 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.
4.11.3. 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 the input clock domain represents the amount of space available in the FIFO (available space = FIFO depth – input fill level).

4.11.4. 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_empty_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.

4.11.5. Single-Clock and Dual-Clock FIFO Core Parameters

<table>
<thead>
<tr>
<th>Table 150. Single-Clock and Dual-Clock FIFO Core Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td>Bits per symbol</td>
</tr>
<tr>
<td>Symbols per beat</td>
</tr>
<tr>
<td>Error width</td>
</tr>
<tr>
<td>FIFO depth</td>
</tr>
<tr>
<td>Use packets</td>
</tr>
<tr>
<td>Channel width</td>
</tr>
</tbody>
</table>

Avalon-ST Single Clock FIFO Only | continued...
### Parameter | Legal Values | Description
---|---|---
Use fill level | — | Turn on this parameter to include the Avalon-MM control and status register interface (CSR). The CSR is enabled when Use fill level is set to 1.

Use Store and Forward | | To turn on Cut-through mode, Use store and forward must be set to 0. Turning on Use store and forward prompts the user to turn on Use fill level, and then the CSR appears.

**Avalon-ST Dual Clock FIFO Only**

Use sink fill level | — | Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.

Use source fill level | — | Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.

Write pointer synchronizer length | 2–8 | The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.

Read pointer synchronizer length | 2–8 | The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.

Use Max Channel | — | Turn on this parameter to specify the maximum channel number.

Max Channel | 1–255 | Maximum channel number.

**Note:** For more information about metastability in Intel devices, refer to *Understanding Metastability in FPGAs*. For more information about metastability analysis and synchronization register chains, refer to the *Managing Metastability*.

**Related Information**

- Managing Metastability with the Software
- Understanding Metastability in FPGAs

### 4.11.6. Avalon-ST Single-Clock FIFO Registers

**Table 151. 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>
</tbody>
</table>

*continued...*
Greater than 0—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the valid signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet.

Note: To turn on Cut-through mode, Use store and forward must be set to 0. Turning on Use store and forward mode prompts the user to turn on Use fill level, and then the CSR appears.

5 drop_on_error RW 0 0—Disables drop-on error.
1—Enables drop-on error.
This register applies only when the Use packet and Use store and forward parameters are turned on.

Table 152. 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 Memory-Mapped Design Optimizations
- Avalon Interface Specifications

4.12. Platform Designer System Design Components Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2017.11.06       | 17.1.0                      | • Changed instances of Qsys to Platform Designer.  
|                  |                             | • Changed instances of AXI Default Slave to Error Response Slave.  
|                  |                             | • Updated topics: Error Response Slave.  
|                  |                             | • Updated Figure: Error Response Slave Parameter Editor.  
|                  |                             | • Added Figure: Error Response Slave Parameter Editor with Enabled CSR Support.  
|                  |                             | • Updated topics: CSR Registers and renamed to Error Response Slave CSR Registers.  
|                  |                             | • Added topic: Error Response Slave Access Violation Service. |
| 2016.05.03       | 16.0.0                      | Updated Address Span Extender  
|                  |                             | • Address Span Extender register mapping better explained  
|                  |                             | • Address Span Extender Parameters table added  
|                  |                             | • Address Span Extender example added |
| 2015.11.02       | 15.1.0                      | Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04       | 15.0.0                      | Avalon-MM Unaligned Burst Expansion Bridge and Avalon-MM Pipeline Bridge, Maximum pending read transactions parameter. Extended description. |
### Related Information

#### Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| December 2014    | 14.1.0                      | • AXI Timeout Bridge.  
|                  |                             | • Added notes to *Avalon-MM Clock Crossing Bridge* pertaining to:  — SDC constraints for its internal asynchronous FIFOs.  — FIFO-based clock crossing. |
| June 2014        | 14.0.0                      | • AXI Bridge support.  
|                  |                             | • Address Span Extender updates.  
|                  |                             | • Avalon-MM Unaligned Burst Expansion Bridge support. |
| November 2013    | 13.1.0                      | • Address Span Extender |
| May 2013         | 13.0.0                      | • Added Streaming Pipeline Stage support.  
|                  |                             | • Added AMBA APB support. |
| November 2012    | 12.1.0                      | • Moved relevant content from the *Embedded Peripherals IP User Guide*. |
5. Creating Platform Designer Components

You can create a Hardware Component Definition File (_hw.tcl) to describe and package IP components for use in a Platform Designer system.

Note: Intel now refers to Qsys as Platform Designer (Standard).

A _hw.tcl describes IP components, interfaces and HDL files. Platform Designer provides the Component Editor to help you create a simple _hw.tcl file.

Refer to the Demo AXI Memory example on the Design Examples page for full code examples that appear in this chapter.

Platform Designer supports Avalon, AMBA 3 AXI (version 1.0), AMBA 4 AXI (version 2.0), AMBA 4 AXI-Lite (version 2.0), AMBA 4 AXI-Stream (version 1.0), and AMBA 3 APB (version 1.0) interface specifications.

Related Information

- Avalon Interface Specifications
- Protocol Specifications
- Demo AXI Memory Example

5.1. Platform Designer Components

A Platform Designer 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 (both or either Synopsys* Design Constraints File .sdc and Intel 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.

5.1.1. Platform Designer Interface Support

Platform Designer is most effective when you use standard interfaces available in the IP Catalog to design custom IP. Standard interfaces operate efficiently with Intel FPGA IP components, and you can take advantage of the bus functional models (BFMs), monitors, and other verification IP that the IP Catalog provides.
Platform Designer supports the following interface specifications:

- Intel FPGA Avalon Memory-Mapped and Streaming
- Arm AMBA 3 AXI (version 1.0)
- Arm AMBA 4 AXI (version 2.0)
- Arm AMBA 4 AXI-Lite (version 2.0)
- Arm AMBA 4 AXI-Stream (version 1.0)
- Arm AMBA 3 APB (version 1.0)

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 Platform Designer system, or export outside of a Platform Designer system.

Platform Designer IP components can include the following interface types:

**Table 153. 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 can be processors and DMAs, while slave memory devices can be RAMs, ROMs, and control registers. Data transfers between master and slave may be uni-directional (read only or write only), or bi-directional (read and write).</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. Platform Designer 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, Platform Designer 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 you export from the Platform Designer system. Platform Designer uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Platform Designer system as a point-to-point connection. Alternatively, you can export conduit interfaces and bring the interfaces 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 Platform Designer system.</td>
</tr>
</tbody>
</table>

**Related Information**

- Avalon Interface Specifications
- AMBA Protocol Specifications
5.1.2. Component Structure

Intel provides components automatically installed with the Intel Quartus Prime software. You can obtain a list of Platform Designer-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 Intel 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 Platform Designer. When you design your own custom component, you can create the _hw.tcl file manually, or by using the Platform Designer 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, Platform Designer 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 filesset 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 312
- [Add Component Instances to a Static or Generated Component](#) on page 315

5.1.3. Component File Organization

A typical component uses the following directory structure where the names of the directories are not significant:
5. Creating Platform Designer Components

<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

Nios II Software Developer’s Handbook
Refer to the "Nios II Software Build Tools" and "Overview of the Hardware Abstraction Layer" chapters.

5.1.4. Component Versions

Platform Designer 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>.

5.1.4.1. Upgrade IP Components to the Latest Version

When you open a Platform Designer design, if Platform Designer 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 Intel 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
In *Intel Quartus Prime Help*
5.2. Design Phases of an IP Component

When you define a component with the Platform Designer Component Editor, or a custom _hw.tcl file, you specify the information that Platform Designer requires to instantiate the component in a Platform Designer system and to generate the appropriate output files for synthesis and simulation.

The following phases describe the process when working with components in Platform Designer:

• **Discovery**—During the discovery phase, Platform Designer 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 Platform Designer, 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, Platform Designer 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 Platform Designer system, the user of the component specifies parameters with the component’s parameter editor.

• **Validation**—During the validation phase, Platform Designer 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, Platform Designer 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, Platform Designer 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.
5.3. Create IP Components in the Platform Designer Component Editor

The Platform Designer Component Editor allows you to create and package an IP component. When you use the Component Editor to define a component, Platform Designer writes the information to an _hw.tcl file.

The Platform Designer 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 Platform Designer 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 Platform Designer 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 Platform Designer 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, Platform Designer 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 Platform Designer 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 Platform Designer Component Editor. The Platform Designer Component Editor overwrites your custom edits.

Example 11. Platform Designer Creates an _hw.tcl File from Entries in the Component Editor

```bash
# 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 463

5.3.1. Save an IP Component and Create the _hw.tcl File

You save a component by clicking Finish in the Platform Designer Component Editor. The Component Editor saves the component as <component_name>_hw.tcl file.

Intel recommends that you move _hw.tcl files and their associated files to an ip/ directory within your Intel 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 Platform Designer for information on how to search for and add components to the IP Catalog for use in your designs.

Related Information
- Creating a System with Platform Designer on page 10
- Publishing Component Information to Embedded Software
  In Nios II Gen 2 Software Developer’s Handbook
5.3.2. Edit an IP Component with the Platform Designer Component Editor

In Platform Designer, 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 overwriting your customized file.

You cannot use the Component Editor to edit components installed with the Intel Quartus Prime software, such as Intel-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.

5.4. Specify IP Component Type Information

The Component Type tab in the Platform Designer 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 Intel Quartus Prime installation directory, the component appears in the specified group under IP Catalog.
- **Description**—Allows you to describe the component. This description appears when the user views the component details.
- **Created By**—Allows you to specify the author of the component.
- **Icon**—Allows you to enter the relative path to an icon file (.gif, .jpg, or .png format) that represents the component and appears as the header in the parameter editor for the component. The default image is the Intel FPGA IP 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:
  - 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 147. 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 Intel Quartus Prime software version that Platform Designer uses to create the _hw.tcl file, and ensures compatibility with this version of the Platform Designer API in future ACDS releases.
Example 12. _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 Intel 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 463

5.5. Create an HDL File in the Platform Designer Component Editor

If you do not have an HDL file for your component, you can use the Platform Designer 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 Platform Designer 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 Platform Designer Component Editor on page 293

5.6. Create an HDL File Using a Template in the Platform Designer Component Editor

You can use a template to create interfaces and signals for your Platform Designer component.
1. In Platform Designer, 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 Platform Designer, right-click your new component in the IP Catalog, and then click **Edit**.

5. In the Platform Designer 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

8. Verify the `<component_name>.v` file appears in the Synthesis Files table on the Files tab.

Related Information

Specify Synthesis and Simulation Files in the Platform Designer Component Editor on page 293

5.7. Specify Synthesis and Simulation Files in the Platform Designer Component Editor

The Files tab in the Platform Designer Component Editor allows you to specify synthesis and simulation files for your custom component.

If you already have an HDL file that describes 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 Intel 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 Platform Designer Component Editor on page 291
- Create an HDL File Using a Template in the Platform Designer Component Editor on page 291

**5.7.1. Specify HDL Files for Synthesis in the Platform Designer Component Editor**

In the Platform Designer 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 Intel Quartus Prime software. The synthesis files for a component are copied to the generation output directory during Platform Designer system generation.
Figure 148. Using HDL Files to Define a Component

In the **Synthesis Files** section on the **Files** tab in the Platform Designer Component Editor, the **demo_axi_memory.sv** file should be selected as the top-level file for the component.

### 5.7.2. Analyze Synthesis Files in the Platform Designer Component Editor

After you specify the top-level HDL file in the Platform Designer 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, Platform Designer 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 13. _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

add_fileset_file single_clk_RAM.v VERILOG PATH single_clk_RAM.v
```
5.7.3. Name HDL Signals for Automatic Interface and Type Recognition in the Platform Designer 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 154. 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>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>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Interface Prefix</th>
<th>Interface Type</th>
</tr>
</thead>
<tbody>
<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
- Protocol Specifications

### 5.7.4. Specify Files for Simulation in the Component Editor

To support Platform Designer 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 Platform Designer Component Editor.

**Note:**
The order that you add files to the fileset determines the order of compilation. For VHDL filessets with VHDL files, you must add the files bottom-up, adding the top-level file last.

**Figure 149. Specifying the Simulation Output Files on the Files Tab**

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** filessets 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 SystemVerilog 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 14. _hw.tcl Created from Entries in the Files tab in the Simulation Files Section

```
add_files SIM_VERILOG SIM_VERILOG "" ""
set_files SIM_VERILOG TOP_LEVEL demo_axi_memory
add_files single_clk_ram.v VERILOG PATH single_clk_ram.v
add_files verbosity_pkg.sv SYSTEM_VERILOG PATH /verification_lib/verbosity_pkg.sv
add_files demo_axi_memory.sv SYSTEM_VERILOG PATH /demo_axi_memory.sv

add_files SIM_VHDL SIM_VHDL "" ""
set_files SIM_VHDL TOP_LEVEL demo_axi_memory
set_files SIM_VHDL ENABLE_RELATIVE_INCLUDE_PATHS false
add_files demo_axi_memory.sv SYSTEM_VERILOG PATH /demo_axi_memory.sv
add_files single_clk_ram.v VERILOG PATH single_clk_ram.v
add_files verbosity_pkg.sv SYSTEM_VERILOG PATH /verification_lib/verbosity_pkg.sv
```

Related Information

Component Interface Tcl Reference on page 463

5.7.5. Include an Internal Register Map Description in the .svd for Slave Interfaces Connected to an HPS Component

Platform Designer 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 15. 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 (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)"
```
5.8. Add Signals and Interfaces in the Platform Designer Component Editor

In the Platform Designer 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 Platform Designer, click **New Component** in the IP Catalog.
2. In the Platform Designer Component Editor, click the **Signals & Interfaces** tab.
3. To add an interface, click **<< add interface >>** in the left pane. A drop-down list appears where you select the interface type.
4. Select an interface from the drop-down list. The selected interface appears in the parameter editor where you can specify its parameters.
5. To add signals for the selected interface click **<< add signal >>** below the selected interface.
6. To move signals between interfaces, select the signal, and then drag it to another interface.
7. To rename a 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, Platform Designer also removes all of its associated signals.
5.9. Specify Parameters in the Platform Designer Component Editor

Components can include parameterized HDL, which allow users of the component flexibility in meeting their system requirements. For example, a component with a configurable memory size or data width, allows using one HDL implementation 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 Platform Designer 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 top-level synthesis file template created from the Files tab include the parameters that you create on the Parameters tab.

When the component includes HDL files, the parameters match those defined in the top-level module, and you cannot 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 create a top-level template HDL file for synthesis with the Component Editor, 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 for 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 editor.

**Figure 151. Parameters Tab in the Platform Designer Components Editor**

On the **Parameters** tab, you can click **Preview the GUI** at any time to see how the declared parameters appear in the parameter editor. Parameters with their default values appear with checks in the **Editable** column. Editable parameters cannot contain computed expressions. You can group parameters under a common heading or section in the 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 16. _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"
set_parameter_property AXI_ID_W HDL_PARAMETER true
add_parameter AXI_ADDRESS_W INTEGER 12
set_parameter_property AXI_ADDRESS_W DEFAULT_VALUE 12
add_parameter AXI_DATA_W INTEGER 32
...
# display items
#
add_display_item "AXI Port Widths" AXI_ID_W PARAMETER ""
```

Note: If an AXI slave's ID bit width is smaller than required for your system, the AXI slave response may not reach all AXI masters. The formula of an AXI slave ID bit width is calculated as follows:

$$\text{maximum_master_id_width_in_the_interconnect} + \log_2(\text{number_of_masters_in_the_same_interconnect})$$

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 155. AXI Master and Slave Parameters

Platform Designer refers to AXI interface parameters to build AXI interconnect. If these parameter settings are incompatible with the component’s HDL behavior, Platform Designer 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 463

5.9.1. Valid Ranges for Parameters in the _hw.tcl File

In the _hw.tcl file, you can specify valid ranges for parameters.
Platform Designer validation checks each parameter value against the ALLOWED_RANGES property. If the values specified are outside of the allowed ranges, Platform Designer 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 156. ALLOWED_RANGES Property

<table>
<thead>
<tr>
<th>ALLOWED_RANGES Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a b c}</td>
<td>a, b, or c</td>
</tr>
<tr>
<td>{&quot;No Control&quot; &quot;Single Control&quot; &quot;Dual Controls&quot;}</td>
<td>Unique string values. Quotation marks are required if the strings include spaces</td>
</tr>
<tr>
<td>{1 2 4 8 16}</td>
<td>1, 2, 4, 8, or 16</td>
</tr>
<tr>
<td>{1:3}</td>
<td>1 through 3, inclusive.</td>
</tr>
<tr>
<td>{1 2 3 7:10}</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 304

### 5.9.2. Types of Platform Designer Parameters

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

Platform Designer User Parameters on page 303
Platform Designer System Information Parameters on page 303
Platform Designer Derived Parameters on page 304

Related Information

Declare Parameters with Custom _hw.tcl Commands on page 304

### 5.9.2.1. Platform Designer 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.

### 5.9.2.2. Platform Designer System Information Parameters

A SYSTEM_INFO parameter is a parameter whose value is set automatically by the Platform Designer 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:

```plaintext
set_parameter_property <param> SYSTEM_INFO CLOCK_RATE
```

You then set the name of the clock interface as the SYSTEM_INFO argument:

```plaintext
set_parameter_property <param> SYSTEM_INFO_ARG <clkname>
```

### 5.9.2.3. Platform Designer 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 304

### 5.9.2.3.1. Parameterized Parameter Widths

Platform Designer 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.

### 5.9.3. 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 17. 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. Platform Designer 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.

```plaintext
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 HDL_PARAMETER true
set_parameter_property AXI_NUMBYTES GROUP "AXI Port Widths"
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"
...
```
5.9.4. 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 \texttt{ALLOWED\_RANGES} property allows. You define a validation callback by setting the \texttt{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 18. 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"
}
```

5.10. Declaring SystemVerilog Interfaces in \_hw.tcl

Platform Designer supports interfaces written in SystemVerilog.
The following example is _hw.tcl for a module with a SystemVerilog interface. The sample code is divided into parts 1 and 2.

Part 1 defines the normal array of parameters, Platform Designer interface, and ports


```tcl
# request TCL package from ACDS 17.1
#
package require -exact qsys 17.1
#
module ram_ip_sv_ifc_hw
#
set_module_property DESCRIPTION ""
set_module_property NAME ram_ip_sv_ifc_hw
set_module_property VERSION 1.0
set_module_property INTERNAL false
set_module_property OPAQUE_ADDRESS_MAP true
set_module_property AUTHOR ""
set_module_property DISPLAY_NAME ram_ip_hw_with_SV_d0
set_module_property INSTANTIATE_IN_SYSTEM_MODULE true
set_module_property EDITABLE true
set_module_property REPORT_TO_TALKBACK false
set_module_property ALLOW_GREYBOX_GENERATION false
set_module_property REPORT_HIERARCHY false

# Part 1 – Add parameter, platform designer interface and ports
# Adding parameter
add_parameter my_interface_parameter STRING "" "I am an interface parameter"

# Adding platform designer interface clk
add_interface clk clock end
set_interface_property clk clockRate 0
# Adding ports to clk interface
add_interface_port clk clk clk Input 1

# Adding platform designer interface reset
add_interface reset reset reset end
set_interface_property reset associatedClock clk
# Adding ports to reset interface
add_interface_port reset reset reset Input 1

# Adding platform designer interface avalon_slave
add_interface avalon_slave avalon end
set_interface_property avalon_slave addressUnits WORDS
# Adding ports to avalon_slave interface
add_interface_port avalon_slave address address Input 10
add_interface_port avalon_slave write write Input 1
add_interface_port avalon_slave readdata readdata Output 32
add_interface_port avalon_slave writedata writedata Input 32
set_interface_property avalon_slave associatedClock clk
set_interface_property avalon_slave associatedReset reset

# Adding ram_ip files
add_filesset synthesis_filesset QUARTUS_SYNTH
set_filesset_property synthesis_filesset TOP_LEVEL ram_ip
add_filesset_file ram_ip.sv SYSTEM_VERILOG PATH ram_ip.sv
```

Part 2 defines the interface name, ports, and parameters of the SystemVerilog interface.
Example 20. Example Part 2: SystemVerilog Interface Parameters in _hw.tcl

```tcl
# Part 2 – Adding SV interface and its properties.
# Adding SV interface
add_sv_interface bus mem_ifc

# Setting the parameter property to add SV interface parameters
set_parameter_property my_interface_parameter SV_INTERFACE_PARAMETER bus

# Setting the port properties to add them to SV interface port
set_port_property clk SV_INTERFACE_PORT bus
set_port_property reset SV_INTERFACE_PORT bus

# Setting the port properties to add them as signals inside SV interface
set_port_property address SV_INTERFACE_SIGNAL bus
set_port_property write SV_INTERFACE_SIGNAL bus
set_port_property writedata SV_INTERFACE_SIGNAL bus
set_port_property readdata SV_INTERFACE_SIGNAL bus

# Adding the SV Interface File
add_fileset_file mem_ifc.sv SYSTEM_VERILOG PATH mem_ifc.sv
```
To instantiate the IP component in your HDL file, click **Generate ➤ Show Instantiation Template** in the Platform Designer editor to display an instantiation template in Verilog or VHDL. Now you can instantiate the IP core in your top-level design HDL file with the template code.

**Figure 153. Instantiation Template Dialog Box**

The following sample contains _hw.tcl to set exportable width values:

**Example 21. Sample _hw.tcl Component with User Alterable Expressions**

```tcl
package require -exact qsys 17.1

set_module_property NAME demo
set_module_property DISPLAY_NAME "Demo"
set_module_property ELABORATION_CALLBACK elaborate

# add exportable hdl parameter RECONFIG_DATA_WIDTH
add_parameter RECONFIG_DATA_WIDTH INTEGER 48
set_parameter_property RECONFIG_DATA_WIDTH AFFECTS_GENERATION false
set_parameter_property RECONFIG_DATA_WIDTH HDL_PARAMETER true

# add exportable hdl parameter RECONFIG_ADDR_WIDTH
add_parameter RECONFIG_ADDR_WIDTH INTEGER 32
set_parameter_property RECONFIG_ADDR_WIDTH AFFECTS_GENERATION false
set_parameter_property RECONFIG_ADDR_WIDTH HDL_PARAMETER true

# add non-exportable hdl parameter
add_parameter l_addr INTEGER 32
set_parameter l_addr HDL_PARAMETER false

# add interface
add_interface s0 conduit end

proc elaborate {} {
    add_interface_port s0 rdata readdata output "reconfig_data_width*2 + l_addr"
    add_interface_port s0 raddr readaddress output [get_parameter_value RECONFIG_ADDR_WIDTH]
    set_port_property raddr WIDTH_EXPR "RECONFIG_ADDR_WIDTH"
}
```
5.12. Scripting Wire-Level Expressions

Platform Designer supports system scripting commands to apply wire-level expressions to input ports in IP components.

The following commands function with the `qsystc-script` utility or in a `_hw.tcl` file to set, retrieve, or remove an expression on a port:

```plaintext
set_wirelevel_expression <instance_or_port_bit> <expression>
get_wirelevel_expressions <instance_or_port_bit>
remove_wirelevel_expressions <instance_or_port_bit>
```

These commands require a string that you compose from the left-handed and right-handed components of the expression. Platform Designer reports errors in syntax, existence, or system hierarchy.

5.13. 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 22. Avalon-ST Source Interface Optionally Included in a Component Specified with an Elaboration Callback**

```plaintext
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_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
- Declare Parameters with Custom _hw.tcl Commands on page 304
- Validate Parameter Values with a Validation Callback on page 306
- Component Interface Tcl Reference on page 463
5.14. 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 23. 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, Platform Designer creates a top-level Platform Designer 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
set_parameter_property 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"
genenerate_my_custom_hdl $csr_enabled demo_axi_memory_gen.sv
add_fileset_file demo_axi_memory_gen.sv VERILOG PATH demo_axi_memory_gen.sv

Related Information
- Specify Synthesis and Simulation Files in the Platform Designer Component Editor on page 293
- Component Interface Tcl Reference on page 463

5.15. 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 Platform Designer interconnect to connect the subsystem instances.

You can add child instances in a composition callback of the _hw.tcl file.

With a composition callback, you can also instantiate and parameterize sub-components as a function of the composed component's parameter values. You define a composition callback by setting the COMPOSITION_CALLBACK module property to the name of the composition callback procedures.

A composition callback replaces the validation and elaboration phases. HDL for the subsystem is generated by generating all of the sub-components and the top-level that combines them.

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

Exporting an interface means that you are making the interface visible from the outside of your component, instead of connecting it internally. You can set the EXPORT_OF property of the externally visible interface from the main program or the composition callback, to indicate that it is an exported view of the submodule's interface.
Exporting an interface is different than defining an interface. An exported interface is an exact copy of the subcomponent’s interface, and you are not allowed to change properties on the exported interface. For example, if the internal interface is a 32-bit or 64-bit master without bursting, then the exported interface is the same. An interface on a subcomponent cannot be exported and also connected within the subsystem.

When you create an exported interface, the properties of the exported interface are copied from the subcomponent’s interface without modification. Ports are copied from the subcomponent’s interface with only one modification; the names of the exported ports on the composed component are chosen to ensure that they are unique.

**Figure 154. Top-Level of a Composed Component**

![Top-Level of a Composed Component Diagram]

**Example 24. Composed _hw.tcl File that Instantiates Two Sub-Components**

Platform Designer 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_interface_property_value reset synchronous_edges deassert
  set_interface_property reset EXPORT_OF reset.in_reset
  set_interface_property slave EXPORT_OF regs.slave
  set_interface_property pins EXPORT_OF phy.pins

  add_connection clk.out_clk reset.clk
  add_connection clk.out_clk regs.clk
  add_connection clk.out_clk phy.clk
  add_connection reset.out_reset regs.reset
  add_connection reset.out_reset phy.clk_reset
```
5.16. Create an IP Component with Platform Designer a System View Different from the Generated Synthesis Output Files

There are cases where it may be beneficial to have the structural Platform Designer 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 Platform Designer system in order to ensure timing closure for their client or team-mate. In this case, the designer creates a design partition for the Platform Designer system, and then exports a post-fit Intel 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 Platform Designer system.

When the client or team-mate adds this component to their Platform Designer system, the designer wants the client or team-mate to see a structural representation of the component, including lower-level components and the address map of the original Platform Designer system. This structural view is a logical representation of the component that is used during the elaboration and validation phases in Platform Designer.

Example 25. Structural Composition Callback and .qxp File as the Generated Output

To specify a structural representation of the component for Platform Designer, connect components or generate a hardware Tcl description of the Platform Designer system, and then insert the Tcl commands into a structural composition callback. To invoke the structural composition callback use the command:

```
set_module_property STRUCTURAL_COMPOSITION_CALLBACK structural_hierarchy
```

```
package require -exact qsys 14.0
set_module_property name example_structural_composition

set_module_property STRUCTURAL_COMPOSITION_CALLBACK \\structural_hierarchy

add_fileset synthesis_fileset QUARTUS_SYNTH \\synth_callback_procedure

add_fileset simulation_fileset SIM_VERILOG \\sim_callback_procedure

set_fileset_property synthesis_fileset TOP_LEVEL \
my_custom_component
```
set_fileset_property simulation_fileset TOP_LEVEL \ my_custom_component

proc structural_hierarchy {} {
    # called during elaboration and validation phase
    # exported ports should be same in structural_hierarchy
    # and generated QXP
    # These commands could come from the exported hardware Tcl
    add_interface clk clock sink
    add_interface reset reset sink
    add_instance clk_0 clock_source
    set_interface_property clk EXPORT_OF clk_0.clk_in
    set_interface_property reset EXPORT_OF clk_0.clk_in_reset
    add_instance pll_0 altera_pll
    # connections and connection parameters
    add_connection clk_0.clk pll_0.refclk clock
    add_connection clk_0.clk_reset pll_0.reset reset
}

proc synth_callback_procedure { entity_name } {
    # the QXP should have the same name for ports
    # as 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 312

5.17. 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, Platform Designer validation-time errors occur, which are often difficult to debug.

**Related Information**
- Static Components on page 316
- Generated Components on page 317

### 5.17.1. 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, Platform Designer generates static IPs only once across multiple instantiations, meaning they have the same top-level name set.

**Example 26. 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 27. Top-Level HDL Instance and Wrapper File Created by Platform Designer**

In this example, Platform Designer 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.
```
5.17.2. 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 Platform Designer 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:

```bash
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:

```bash
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 28. Typical Usage of the add_hdl_instance Command for Generated Components**

Platform Designer generates a wrapper file for the instance name specified in the `_hw.tcl` file.

```bash
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 Platform Designer 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
    set autogeneratedfixedname [get_instance_property emif_instance_name HDLINSTANCE_GET_GENERATED_NAME]
    set fileID [open "generated_toplevel_component.v" r]
    set temp ""
    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}"
        # replace the autogenerated name for emif_instance_name added
        regsub -all "substitute_autogenerated_emifinstancename_here" $lineInfo "${ autogeneratedfixedname }"
        # via add_hdl_instance
        regsub -all "$autogeneratedfixedname" $lineInfo "${entity_name}"
        regsub -all "substitute_autogenerated_emifinstancename_here" $lineInfo "${entity_name}"
    }
}
```
Example 29. Top-Level HDL Instance and Wrapper File Created By Platform Designer

```vhdl
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
// Generated using ACDS version 13.
`timescale 1 ps / 1 ps
module generated_toplevel_component_0_emif_instance_name (  
  input wire pll_ref_clk, // pll_ref_clk.clk
  input wire global_reset_n, // global_reset.reset_n
  input wire soft_reset_n, // soft_reset.reset_n
  output wire afi_clk, // afi_clk.clk
  ...
  ...);
example_addhdlinstance_system_add_hdl_instance_example_0_emif
_instance_name_emif_instance_name emif_instance_name (  
  .pll_ref_clk (pll_ref_clk), // pll_ref_clk.clk
  .global_reset_n (global_reset_n), // global_reset.reset_n
  .soft_reset_n (soft_reset_n), // soft_reset.reset_n
  ...
  ...);
endmodule
```

Related Information

- Control File Generation Dynamically with Parameters and a Fileset Callback on page 311
- Intellectual Property & Reference Designs
5.17.3. Design Guidelines for Adding Component Instances

In order to promote standard and predictable results when generating static and generated components, Intel 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.

5.18. Creating Platform Designer Components Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018.05.07</td>
<td>18.0</td>
<td>• Added scripting support for wire-level expressions.</td>
</tr>
</tbody>
</table>
| 2017.11.06        | 17.1.0                     | • Changed instances of Qsys to Platform Designer (Standard)  
  • Replaced mentions of altera_axi_default_slave to altera_error_response_slave |
| 2017.05.08        | 17.0.0                     | • Updated Figure: Address Span Extender |
| 2015.11.02        | 15.1.0                     | • Updated screen shots Files tab, Qsys Component Editor.  
  • Added topic: Specify Interfaces and Signals in the Qsys Component Editor.  
  • Added topic: Create an HDL File in the Qsys Component Editor.  
  • Added topic: Create an HDL File Using a Template in the Qsys Component Editor. |
| 2015.05.04        | 15.0.0                     | • add_hdl_instance  
  • Added Creating a Component With Differing Structural Qsys View and Generated Output Files. |
| November 2013     | 13.1.0                     | • Consolidated content from other Qsys chapters.  
  • Added Upgrading IP Components to the Latest Version.  
  • Updated for AMBA APB support. |
<p>| May 2013          | 13.0.0                     |     |</p>
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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 AXI 3 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**

**Documentation Archive**

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.
6. Platform Designer Command-Line Utilities

You can perform many of the functions available in the Platform Designer GUI at the command-line, with Platform Designer command-line utilities.

You run Platform Designer command-line executables from the Intel Quartus Prime installation directory:

```
<Intel Quartus Prime installation directory>/quartus\sopc_builder \bin
```

For command-line help listing of all the options for any executable, type the following command:

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

**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 utility from any directory location.

6.1. Run the Platform Designer Editor with qsys-edit

The `qsys-edit` utility allows you to run the Platform Designer editor from command-line.

You can use the following options with the `qsys-edit` utility:

**Table 157. 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>Specifies the name of the .qsys system or .qvar variation file to edit.</td>
</tr>
<tr>
<td><code>--search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>If you omit this command, Platform Designer uses a standard default path. If you provide a search path, Platform Designer searches a comma-separated list of paths. To include the standard path in your replacement, use &quot;$&quot;, for example: <code>/extra/dir,$</code>.</td>
</tr>
<tr>
<td><code>--family=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Sets the device family.</td>
</tr>
<tr>
<td><code>--part=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Sets the device part number. If set, this option overrides the --family option.</td>
</tr>
</tbody>
</table>
## 6. Platform Designer Command-Line Utilities

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--project-directory[=&lt;directory&gt;]</td>
<td>Optional</td>
<td>Specifies the component locations relative to the project, if any. Default option is <code>.</code> (current directory). To exclude any project directory, use <code>''</code>.</td>
</tr>
<tr>
<td>--new-component-type[=&lt;value&gt;]</td>
<td>Optional</td>
<td>Specifies the instance type for parameterization in a variation.</td>
</tr>
<tr>
<td>--system-info[=&lt;DEVICE_FAMILY</td>
<td>DEVICE_FEATURES</td>
<td>CLOCK_RATE] \</td>
</tr>
<tr>
<td>--require-generation</td>
<td>Optional</td>
<td>Marks the loading system as requiring generation.</td>
</tr>
<tr>
<td>--debug</td>
<td>Optional</td>
<td>Enables debugging features and output.</td>
</tr>
<tr>
<td>--host-controller</td>
<td>Optional</td>
<td>Specifies the instance type that you want to parameterize in a variation.</td>
</tr>
<tr>
<td>--jvm-max-heap-size=&lt;value&gt;</td>
<td>Optional</td>
<td>The maximum memory size that Platform Designer uses when running qsys-edit. You specify this value as <code>&lt;size&gt;&lt;unit&gt;</code>, where unit is <code>m</code> (or <code>M</code>) for multiples of megabytes, or <code>g</code> (or <code>G</code>) for multiples of gigabytes. The default value is 512m.</td>
</tr>
<tr>
<td>--help</td>
<td>Optional</td>
<td>Displays help for qsys-edit.</td>
</tr>
</tbody>
</table>

**Important:** The options **--quartus-project** and **--new-quartus-project** are mutually exclusive. If you use **--quartus-project** you cannot use **--new-quartus-project** and vice versa.

**Extended Features with the --debug Options**

The **--debug** option provides powerful tools for debugging. When you launch Platform Designer with the **--debug** option enabled, you can:

- View debug messages when opening a system or generating HDL for that system.
- Add the **--verbose** argument when generating IP or a system using command-line utilities.
- Access internal library components in the IP Catalog, for example, modules used to create interconnect fabric.
- Access to debug tools and files from the Internal menu.
### Table 158. Debug Options on the Internal Menu

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show hw.tcl Debugger</td>
<td>Displays a Tcl debugger.</td>
</tr>
<tr>
<td>Show System File</td>
<td>Displays the current system XML in a text dialog box.</td>
</tr>
<tr>
<td>Show SOPCINFO File</td>
<td>Shows the SOPCINFO report XML in a text dialog box.</td>
</tr>
<tr>
<td>Show UI Properties</td>
<td>Displays the UI properties in a text dialog box.</td>
</tr>
<tr>
<td>Show Command Line Arguments</td>
<td>Displays all command-line arguments and environment variables in a text dialog box.</td>
</tr>
<tr>
<td>Show System Changes</td>
<td>Displays dynamic system changes in a text dialog box.</td>
</tr>
<tr>
<td>Make Model Read-only</td>
<td>Makes the system you are working in read-only.</td>
</tr>
<tr>
<td>Take Screenshots</td>
<td>Creates a .png file in the <code>&lt;project_directory&gt;</code> by default. You can navigate and save to a directory of your choice.</td>
</tr>
<tr>
<td>Show Plug-In Catalog</td>
<td>Displays library details such as type, version, tags, etc. for all IPs in the IP Catalog.</td>
</tr>
<tr>
<td>Show Adapter Reports</td>
<td>Displays adapter reports for any adapters added when transforming the system.</td>
</tr>
</tbody>
</table>

- You can view detailed debugging messages in the **Component Editor** while building a custom IP component.
- You can view the generated Tcl script while editing in the **Component Editor** with the **Advanced ➤ Show Tcl for Component** command.
- You can launch the System Console with debug logging.

### 6.2. Scripting IP Core Generation

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

To parameterize and generate an IP core at command-line, follow these steps:
1. Run `qsys-script` to start a Tcl script that instantiates the IP and sets parameters:

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

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

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

**Related Information**

Generate a Platform Designer System with `qsys-script` on page 329

### 6.2.1. qsys-generate Command-Line Options

Table 159. Command-Line Options for `qsys-generate`

Table in alphabetical order.

<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>Specifies the name of the <code>.qsys</code> system file to generate.</td>
</tr>
<tr>
<td><code>--block-symbol-file</code></td>
<td>Optional</td>
<td>Creates a Block Symbol File (.bsf) for the Platform Designer system.</td>
</tr>
<tr>
<td><code>--clear-output-directory</code></td>
<td>Optional</td>
<td>Clears the output directory corresponding to the selected target, that is, simulation or synthesis.</td>
</tr>
<tr>
<td><code>--example-design=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Creates example design files. For example, <code>--example-design</code> or <code>--example-design=all</code>. The default is <code>All</code>, which generates example designs for all instances. Alternatively, choose specific filesets based on instance name and fileset name. For example <code>--example-design=instance0.example_design1,instance1.example_design2</code>. Specify an output directory for the example design files creation.</td>
</tr>
<tr>
<td><code>--export-qsys-script</code></td>
<td>Optional</td>
<td>If you set this option to true, Platform Designer exports the post-generation system as a Platform Designer script file with the extension <code>.tcl</code>.</td>
</tr>
<tr>
<td><code>--family=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Sets the device family name.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Optional</td>
<td>Displays help for <code>--qsys-generate</code>.</td>
</tr>
<tr>
<td><code>--greybox</code></td>
<td>Optional</td>
<td>If you are synthesizing your design with a third-party EDA synthesis tool, generate a netlist for the synthesis tool to estimate timing and resource usage for this design.</td>
</tr>
<tr>
<td><code>--ipxact</code></td>
<td>Optional</td>
<td>If you specify this option, Platform Designer generates the post-generation system as an IPXACT-compatible component description. Note: Platform Designer supports importing and exporting files in IP-XACT 2009 format and exporting IP-XACT files in 2014 format.</td>
</tr>
<tr>
<td><code>--jvm-max-heap-size=&lt;value&gt;</code></td>
<td>Optional</td>
<td>The maximum memory size that Platform Designer uses 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>
</tbody>
</table>

continued...
### Option Usage Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--parallel[=&lt;level&gt;]</code></td>
<td>Optional</td>
<td>Directs Platform Designer to generate in parallel mode, with the level of parallelism that you specify. If you omit the level, Platform Designer determines a number based on processor availability and number of files to be generated.</td>
</tr>
<tr>
<td><code>--part=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Sets the device part number. If set, this option overrides the <code>--family</code> option.</td>
</tr>
<tr>
<td><code>--output-directory=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Sets the output directory. Platform Designer creates each generation target in a sub-directory of the output directory. If you do not specify the output directory, Platform Designer uses a sub-directory of the current working directory matching the name of the system.</td>
</tr>
<tr>
<td><code>--search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>If you omit this command, Platform Designer uses a standard default path. If you provide this command, Platform Designer 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>`--simulation=&lt;VERILOG</td>
<td>VHDL&gt;`</td>
<td>Optional</td>
</tr>
<tr>
<td>`--synthesis=&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><code>--upgrade-ip-cores</code></td>
<td>Optional</td>
<td>Enables upgrading all the IP cores that support upgrade in the Platform Designer system.</td>
</tr>
<tr>
<td><code>--upgrade-variation-file</code></td>
<td>Optional</td>
<td>If you set this option to true, the file argument for this command accepts a .v file, which contains a IP variant. This file parameterizes a corresponding instance in a Platform Designer system of the same name.</td>
</tr>
</tbody>
</table>

### 6.3. Display Available IP Components with ip-catalog

The `ip-catalog` command displays a list of available IP components relative to the current Intel Quartus Prime project directory, as either text or XML.
You can use the following options with the **ip-catalog** utility:

### Table 160.  ip-catalog Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--project-dir= &lt;directory&gt;</td>
<td>Optional</td>
<td>Finds IP components relative to the Intel Quartus Prime project directory. By default, Platform Designer uses '.' as the current directory. To exclude a project directory, leave the value empty.</td>
</tr>
<tr>
<td>--type</td>
<td>Optional</td>
<td>Provides a pattern to filter the type of available plug-ins. By default, Platform Designer shows only IP components. To look for a partial type string, surround with '*', for instance, <em>connection</em>.</td>
</tr>
<tr>
<td>--name= &lt;value&gt;</td>
<td>Optional</td>
<td>Provides a pattern to filter the names of the IP components found. To show all IP components, use a '<em>' or '. '. By default, Platform Designer shows all IP components. The argument is not case sensitive. To look for a partial name, surround with '</em>', for instance, <em>uart</em></td>
</tr>
<tr>
<td>--verbose</td>
<td>Optional</td>
<td>Reports the progress of the command.</td>
</tr>
<tr>
<td>--xml</td>
<td>Optional</td>
<td>Generates the output in XML format, in place of colon-delimited format.</td>
</tr>
<tr>
<td>--search-path= &lt;value&gt;</td>
<td>Optional</td>
<td>If you omit this command, Platform Designer uses a standard default path. If you provide this command, Platform Designer 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>&lt;1st arg value&gt;</td>
<td>Optional</td>
<td>Specifies the directory or name fragment.</td>
</tr>
<tr>
<td>--jvm-max-heap-size= &lt;value&gt;</td>
<td>Optional</td>
<td>The maximum memory size that Platform Designer uses for when running ip-catalog. 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>
<tr>
<td>--help</td>
<td>Optional</td>
<td>Displays help for the ip-catalog command.</td>
</tr>
</tbody>
</table>

### 6.4. Create an .ipx File with ip-make-ipx

The **ip-make-ipx** command creates an .ipx index file. This file provides 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.

You can use the following options with the **ip-make-ipx** utility:

### Table 161.  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 directory containing your IP components. The default directory is '. '. 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 index file to generate. The default name is /component.ipx. Set as --output='' to print the output to the console.</td>
</tr>
</tbody>
</table>

*continued...*
### 6.5. Generate Simulation Scripts

You can use the `ip-make-simscript` utility to generate simulation scripts for one or more simulators, given one or more Simulation Package Descriptor (.spd) files, .qsys files, and .ip files.

In Platform Designer, `ip-make-simscript` generates simulation scripts in a hierarchical structure instead of a flat view of the entire system. The `ip-make-simscript` utility uses .spd and system files according to the options you select:

- **When targeting only .spd files** (`ip-make-simscript --spd=<file>.spd`) the utility combines the contents of all input .spd files, and generates a common directory which contains a set of `<simulator>_files.tcl` files under the specified output directory.

- **When targeting only system files** (`ip-make-simscript --system-file=<file>`) such as .qsys and .ip files, the utility searches for instances of `<simulator>_files.tcl` files for each input system, and generates a combined simulation script which contains a list of references of `<simulator>_files.tcl`.

- **When the utility uses both --spd and --system-file options**, `ip-make-simscript` combines all input .spd files and generates a common/ `<simulator>_files.tcl` in the specified output directory. The generated simulation script refers to the generated `common/<simulator>_files.tcl` first, followed by a list of Tcl files from each input system.

#### Table 162. ip-make-simscript Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--spd=&lt;file&gt;</td>
<td>Optional/Repeatable</td>
<td>The .spd files describe the list of HDL files for simulation, and memory models hierarchy. This argument can either be a single path to an .spd file or a comma-separated list of paths of .spd files. For instance, <code>--spd=ipcore_1.spd,ipcore_2.spd</code> The generated list is processed in the order of the input .spd files.</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 or variables wherever possible. You can specify multiple variables as a comma-separated list.</td>
</tr>
<tr>
<td>--thorough-descent</td>
<td>Optional</td>
<td>If you set this option, Platform Designer searches all the component files, without skipping the sub-directories.</td>
</tr>
<tr>
<td>--message-before=&lt;value&gt;</td>
<td>Optional</td>
<td>Prints a log message at the start of reading an index file.</td>
</tr>
<tr>
<td>--message-after=&lt;value&gt;</td>
<td>Optional</td>
<td>Prints a log message at the end of reading an index file.</td>
</tr>
<tr>
<td>--jvm-max-heap-size=&lt;value&gt;</td>
<td>Optional</td>
<td>The maximum memory size Platform Designer uses when running ipr-make-ipx. You specify this value as <code>&lt;size&gt;&lt;unit&gt;</code>, where unit is <code>m</code> (or <code>M</code>) for multiples of megabytes, or <code>g</code> (or <code>G</code>) for multiples of gigabytes. The default value is <code>512m</code>.</td>
</tr>
<tr>
<td>--help</td>
<td>Optional</td>
<td>Displays help for the <code>ip-make-ipx</code> command.</td>
</tr>
</tbody>
</table>
### Option Table

<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/Repeatable</td>
<td>Specifies the system files (.qsys or .ip files) used to generate the simulation scripts. This argument can contain either a single path to a Platform Designer system file or a comma-separated list of paths to Platform Designer system files. The simulation script is generated in the order the system files are listed. Note: When this argument is used in combination with --spd, the .spd files are parsed before the system files.</td>
</tr>
<tr>
<td><code>-output-directory=&lt;directory&gt;</code></td>
<td>Optional</td>
<td>Specifies the directory path for the location of output files. If you do not specify a directory, the output directory defaults to the directory from which <code>ip-make-simscript</code> runs.</td>
</tr>
<tr>
<td><code>-compile-to-work</code></td>
<td>Optional</td>
<td>Compiles all design files to the default library - <code>work</code>.</td>
</tr>
<tr>
<td><code>-use-relative-paths</code></td>
<td>Optional</td>
<td>Uses relative paths whenever possible.</td>
</tr>
<tr>
<td><code>-nativelink-mode</code></td>
<td>Optional</td>
<td>Generates files for Intel Quartus Prime NativeLink RTL simulation.</td>
</tr>
<tr>
<td><code>-cache-file=&lt;file&gt;</code></td>
<td>Optional</td>
<td>Generates cache file for managed flow.</td>
</tr>
<tr>
<td><code>-quiet</code></td>
<td>Optional</td>
<td>Quiet reporting mode. Does not report generated files.</td>
</tr>
<tr>
<td><code>-jvm-max-heap-size=&lt;value&gt;</code></td>
<td>Optional</td>
<td>The maximum memory size Platform Designer uses when running <code>ip-make-simscript</code>. 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>-search-path=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Comma-separated list of search paths. If omitted, a default path including the current working directory is used. To include the standard path in your replacement, append the $ symbol, for example: &quot;/extra/dir,$&quot;</td>
</tr>
<tr>
<td><code>-device-family=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Overrides the existing device family when used.</td>
</tr>
<tr>
<td><code>-top-name=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Specify a top-level entity name used in generated simulation scripts.</td>
</tr>
<tr>
<td><code>-help</code></td>
<td>Optional</td>
<td>Displays help for <code>--ip-make-simscript</code>.</td>
</tr>
</tbody>
</table>

### 6.6. Generate a Platform Designer System with qsys-script

You can use the `qsys-script` utility to create and manipulate a Platform Designer system with Tcl scripting commands. If you specify a system, Platform Designer loads that system before executing any of the 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.

**Example 30. Platform Designer Command-Line Scripting**

```
qsys-script --script=my_script.tcl \
--system-file=fancy.qsys
```

`my_script.tcl` contains:

```tcl
package require -exact qsys 16.0
# get all instance names in the system and print one by one
set instances [ get_instances ]
foreach instance $instances {
    send_message Info "$instance"
}
```

You can use the following options with the `qsys-script` utility:

**Table 163. 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. Platform Designer 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 a Platform Designer system. If you specify both <code>--cmd</code> and <code>--script</code>, Platform Designer 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 Platform Designer system. If you specify both <code>--cmd</code> and <code>--script</code>, Platform Designer 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 Intel Quartus Prime software supports Tcl API scripting commands. The minimum supported version is 12.0. 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, a Platform Designer uses a standard default path. If you provide this command, Platform Designer searches a comma-separated list of paths. To include the standard path in your replacement, use <code>*s*</code>, for example, <code>/&lt;directory path&gt;/dir,$s</code>. Separate multiple directory references with a comma.</td>
</tr>
<tr>
<td><code>--quartus-project=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Specifies the path to a <code>.qpf</code> Intel Quartus Prime project file. Utilizes the specified Intel Quartus Prime project to add the file saved using save_system command. If you omit this command, Platform Designer uses the default revision as the project name.</td>
</tr>
<tr>
<td><code>--new-quartus-project=&lt;value&gt;</code></td>
<td>Optional</td>
<td>Specifies the name of the new Intel Quartus Prime project. Creates a new Intel Quartus Prime project at the specified path and adds the file saved using save_system command to the project. If you omit this command, Platform Designer uses the Intel Quartus Prime project revision as the new Intel Quartus Prime project name.</td>
</tr>
</tbody>
</table>

...continued...
### Option Table

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--rev=&lt;value&gt;</td>
<td>Optional</td>
<td>Allows you to specify the name of the Intel Quartus Prime project revision.</td>
</tr>
<tr>
<td>--jvm-max-heap-size=&lt;value&gt;</td>
<td>Optional</td>
<td>The maximum memory size that the qsys-script tool uses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specify this value as <code>&lt;size&gt;&lt;unit&gt;</code>, where unit is m (or M)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for multiples of megabytes, or g (or G) for multiples of gigabytes.</td>
</tr>
<tr>
<td>--help</td>
<td>Optional</td>
<td>Displays help for the qsys-script utility.</td>
</tr>
</tbody>
</table>

### Related Information

Intel FPGA Wiki: Platform Designer Scripts

### 6.7. Platform Designer Scripting Command Reference

Platform Designer system scripting provides Tcl commands to manipulate your system. The `qsys-script` provides a command-line alternative to the Platform Designer tool. Use the `qsys-script` commands to create and modify your system, as well as to create reports about the system.

To use the current version of the Tcl commands, include the following line at the top of your script:

```tcl
package require -exact qsys <version>
```

For example, for the current release of the Intel Quartus Prime software, include:

```tcl
package require -exact qsys 18.0
```

The Platform Designer scripting commands fall under the following categories:

- **System** on page 332
- **Subsystems** on page 345
- **Instances** on page 354
- **Connections** on page 387
- **Top-level Exports** on page 399
- **Validation** on page 412
- **Miscellaneous** on page 418
- **Wire-Level Connection Commands** on page 431
6.7.1. System

This section lists the commands that allow you to manipulate a Platform Designer system.

- `create_system` on page 333
- `export_hw_tcl` on page 334
- `get_device_families` on page 335
- `get_devices` on page 336
- `get_module_properties` on page 337
- `get_module_property` on page 338
- `get_project_properties` on page 339
- `get_project_property` on page 340
- `load_system` on page 341
- `save_system` on page 342
- `set_module_property` on page 45
- `set_project_property` on page 344
6.7.1.1. create_system

**Description**
Replaces the current system with a new system of the specified name.

**Usage**
create_system [<name>]

**Returns**
No return value.

**Arguments**

*name (optional)*  The new system name.

**Example**
create_system my_new_system_name

**Related Information**
- load_system on page 341
- save_system on page 342
6.7.1.2. export_hw_tcl

**Description**

Allows you to save the currently open system as an `_hw.tcl` file in the project directory. The saved systems appears under the `System` category in the IP Catalog.

**Usage**

`export_hw_tcl`

**Returns**

No return value.

**Arguments**

No arguments

**Example**

`export_hw_tcl`

**Related Information**

- `load_system` on page 341
- `save_system` on page 342
6.7.1.3. get_device_families

**Description**
Returns the list of installed device families.

**Usage**
get_device_families

**Returns**

*String[]*   The list of device families.

**Arguments**
No arguments

**Example**

```plaintext
get_device_families
```

**Related Information**

*get_devices* on page 336
6.7.1.4. get_devices

**Description**
Returns the list of installed devices for the specified family.

**Usage**

```
get_devices <family>
```

**Returns**

`String[]` The list of devices.

**Arguments**

`family` Specifies the family name to get the devices for.

**Example**

```
get_devices exampleFamily
```

**Related Information**

`get_device_families` on page 335
6.7.1.5. get_module_properties

**Description**
Returns the properties that you can manage for a top-level module of the Platform Designer system.

**Usage**
get_module_properties

**Returns**
The list of property names.

**Arguments**
No arguments.

**Example**

```
get_module_properties
```

**Related Information**
- get_module_property on page 338
- set_module_property on page 45
6.7.1.6. get_module_property

**Description**
Returns the value of a top-level system property.

**Usage**
get_module_property <property>

**Returns**
The property value.

**Arguments**

- *property*  The property name to query. Refer to *Module Properties*.

**Example**

get_module_property NAME

**Related Information**
- get_module_properties on page 337
- set_module_property on page 45
6.7.1.7. get_project_properties

Description
Returns the list of properties that you can query for properties pertaining to the Intel Quartus Prime project.

Usage
get_project_properties

Returns
The list of project properties.

Arguments
No arguments

Example
get_project_properties

Related Information
- get_project_property on page 340
- set_project_property on page 344
6.7.1.8. **get_project_property**

**Description**

Returns the value of an Intel Quartus Prime project property.

**Usage**

`get_project_property <property>`

**Returns**

The property value.

**Arguments**

- `property`  The project property name. Refer to *Project properties*.

**Example**

`get_project_property DEVICE_FAMILY`

**Related Information**

- `get_module_properties` on page 337
- `get_module_property` on page 338
- `set_module_property` on page 45
- *Project Properties* on page 449
6.7.1.9. load_system

**Description**
Loads the Platform Designer 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 the `.qsys` file.

**Example**
load_system example.qsys

**Related Information**
- [create_system](#) on page 333
- [save_system](#) on page 342
6.7.1.10. save_system

Description
Saves the current system to the specified file. If you do not specify the file, Platform Designer saves the system to the same file opened with the load_system command.

Usage
save_system <file>

Returns
No return value.

Arguments
<file>  If available, the path of the .qsys file to save.

Example

<table>
<thead>
<tr>
<th>save_system</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_system file.qsys</td>
</tr>
</tbody>
</table>

Related Information
- load_system on page 341
- create_system on page 333
6.7.1.11. set_module_property

**Description**
Specifies the Tcl procedure to evaluate changes in Platform Designer system instance parameters.

**Usage**
set_module_property <property> <value>

**Returns**
No return value.

**Arguments**

property  The property name. Refer to Module Properties.

value  The new value of the property.

**Example**
set_module_property COMPOSITION_CALLBACK "my_composition_callback"

**Related Information**
- get_module_properties on page 337
- get_module_property on page 338
- Module Properties on page 443
6.7.1.12. set_project_property

**Description**
Sets the project property value, such as the device family.

**Usage**

```
set_project_property <property> <value>
```

**Returns**
No return value.

**Arguments**

- **property**  The property name. 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 339
- get_project_property on page 340
- Project Properties on page 449
6. Platform Designer Command-Line Utilities

6.7.2. Subsystems

This section lists the commands that allow you to obtain the connection and parameter information of instances in your Platform Designer subsystem.

get_composed_connections on page 346
get_composed_connection_parameter_value on page 347
get_composed_connection_parameters on page 348
get_composed_instance_assignment on page 349
get_composed_instance_assignments on page 350
get_composed_instance_parameter_value on page 351
get_composed_instance_parameters on page 352
get_composed_instances on page 353
6.7.2.1. get_composed_connections

**Description**
Returns the list of all connections in the subsystem for an instance that contains the subsystem of the Platform Designer system.

**Usage**
```
get_composed_connections <instance>
```

**Returns**
The list of connection names in the subsystem.

**Arguments**

*instance*  The child instance containing the subsystem.

**Example**
```
get_composed_connections subsystem_0
```

**Related Information**
- [get_composed_connection_parameter_value](#) on page 347
- [get_composed_connection_parameters](#) on page 348
6.7.2.2. get_composed_connection_parameter_value

Description
Returns the parameter value of a connection in a child instance containing the subsystem.

Usage
get_composed_connection_parameter_value <instance> <child_connection> <parameter>

Returns
The parameter value.

Arguments

instance  The child instance that contains the subsystem.

child_connection  The connection name in the subsystem.

parameter  The parameter name to query for the connection.

Example

get_composed_connection_parameter_value subsystem_0 cpu.data_master/memory.s0 baseAddress

Related Information

• get_composed_connection_parameters on page 348
• get_composed_connections on page 346
6.7.2.3. get_composed_connection_parameters

**Description**
Returns the list of parameters of a connection in the subsystem, for an instance that contains the subsystem.

**Usage**
ge_
\text{get\_composed\_connection\_parameters} \ <\text{instance}> \ <\text{child\_connection}>

**Returns**
The list of parameter names.

**Arguments**

\text{instance} \quad \text{The child instance containing the subsystem.}

\text{child\_connection} \quad \text{The name of the connection in the subsystem.}

**Example**
ge_
\text{get\_composed\_connection\_parameters} \ \text{subsystem\_0 cpu.data\_master/memory.s0}

**Related Information**
- \text{get\_composed\_connection\_parameter\_value} on page 347
- \text{get\_composed\_connections} on page 346
6.7.2.4. get_composed_instance_assignment

**Description**
Returns the assignment value of the child instance in the subsystem.

**Usage**
get_composed_instance_assignment <instance> <child_instance> <assignment>

**Returns**
The assignment value.

**Arguments**
- *instance*  The subsystem containing the child instance.
- *child_instance*  The child instance name 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 350
- [get_composed_instances](#) on page 353
6.7.2.5. get_composed_instance_assignments

**Description**
Returns the list of assignments of the child instance in the subsystem.

**Usage**
get_composed_instance_assignments <instance> <child_instance>

**Returns**
The list of assignment names.

**Arguments**

*instance*   The subsystem containing the child instance.

*child_instance*   The child instance name in the subsystem.

**Example**
get_composed_instance_assignments subsystem_0 cpu

**Related Information**
- [get_composed_instance_assignment](#) on page 349
- [get_composed_instances](#) on page 353
6.7.2.6. get_composed_instance_parameter_value

**Description**
Returns the parameter value of the child instance in the subsystem.

**Usage**
get_composed_instance_parameter_value <instance> <child_instance> <parameter>

**Returns**
The parameter value of the instance in the subsystem.

**Arguments**

*instance*  The subsystem containing the child instance.

*child_instance*  The child instance name in the subsystem.

*parameter*  The parameter name to query on the child instance in the subsystem.

**Example**
get_composed_instance_parameter_value subsystem_0 cpu DATA_WIDTH

**Related Information**
- get_composed_instance_parameters on page 352
- get_composed_instances on page 353
6.7.2.7. get_composed_instance_parameters

Description
Returns the list of parameters of the child instance in the subsystem.

Usage
get_composed_instance_parameters <instance> <child_instance>

Returns
The list of parameter names.

Arguments

instance  The subsystem containing the child instance.

child_instance  The child instance name in the subsystem.

Example
get_composed_instance_parameters subsystem_0 cpu

Related Information
• get_composed_instance_parameter_value on page 351
• get_composed_instances on page 353
6.7.2.8. get_composed_instances

**Description**
Returns the list of child instances in the subsystem.

**Usage**
geet_composed_instances <instance>

**Returns**
The list of instance names in the subsystem.

**Arguments**

*instance*  The subsystem containing the child instance.

**Example**
geet_composed_instances subsystem_0

**Related Information**
- get_composed_instance_assignment on page 349
- get_composed_instance_assignments on page 350
- get_composed_instance_parameter_value on page 351
- get_composed_instance_parameters on page 352
6.7.3. Instances

This section lists the commands that allow you to manipulate the instances of IP components in your Platform Designer system.

- **add_instance** on page 355
- **apply_instance_preset** on page 356
- **create_ip** on page 357
- **add_component** on page 358
- **duplicate_instance** on page 359
- **enable_instance_parameter_update_callback** on page 360
- **get_instance_assignment** on page 361
- **get_instance_assignments** on page 362
- **get_instance_documentation_links** on page 363
- **get_instance_interface_assignment** on page 364
- **get_instance_interface_assignments** on page 365
- **get_instance_interface_parameter_property** on page 366
- **get_instance_interface_parameter_value** on page 367
- **get_instance_interface_parameters** on page 368
- **get_instance_interface_port_property** on page 369
- **get_instance_interface_ports** on page 370
- **get_instance_interface_properties** on page 371
- **get_instance_interfaces** on page 372
- **get_instance_parameter_property** on page 373
- **get_instance_parameter_value** on page 374
- **get_instance_parameter_values** on page 375
- **get_instance_parameters** on page 376
- **get_instance_port_property** on page 377
- **get_instance_properties** on page 378
- **get_instance_property** on page 379
- **get_instances** on page 380
- **is_instance_parameter_update_callback_enabled** on page 381
- **remove_instance** on page 382
- **set_instance_parameter_value** on page 383
- **set_instance_parameter_values** on page 384
- **set_instance_property** on page 385
- **set_instance_property** on page 386
6.7.3.1. 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. Platform Designer 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 you do not specify any instance, Platform Designer uses the latest version.

Example
add_instance uart_0 altera_avalon_uart 16.1

Related Information

- get_instance_property on page 380
- get_instances on page 381
- remove_instance on page 383
- set_instance_parameter_value on page 384
- get_instance_parameter_value on page 39
6.7.3.2. apply_instance_preset

**Description**
Applies the settings in a preset to the specified instance.

**Usage**
apply_instance_preset <preset_name>

**Returns**
No return value.

**Arguments**

*preset_name*  The preset name.

**Example**
apply_preset "Custom Debug Settings"

**Related Information**
set_instance_parameter_value on page 384
6.7.3.3. create_ip

**Description**
Creates a new IP Variation system with the given instance.

**Usage**
create_ip <type> [ <instance_name> <version>]

**Returns**
No return value.

**Arguments**

*type* Kind of instance available in the IP catalog, for example, 
altera_avalon_uart.

*instance_name* (optional) A unique local name that you can use to manipulate the instance. If not specified, Platform Designer uses a default name.

*version* (optional) The required version of the specified instance type. If not specified, Platform Designer uses the latest version.

**Example**
create_ip altera_avalon_uart altera_avalon_uart_inst 17.0

**Related Information**
- add_component on page 358
- load_system on page 341
- save_system on page 342
- set_instance_parameter_value on page 384
6.7.3.4. add_component

**Description**
Adds a new IP Variation component to the system.

**Usage**
```
add_component <instance_name> <file_name> [<component_type>
<component_instance_name> <component_version>]
```

**Returns**
No return value.

**Arguments**

`instance_name` A unique local name that you can use to manipulate the instance.

`file_name` The IP variation file name. If a path is not specified, Platform Designer saves the file in the ./ip/system/ sub-folder of your system.

`component_type` *(optional)* The kind of instance available in the IP catalog, for example altera_avalon_uart.

`component_instance_name` *(optional)* The instance name of the component in the IP variation file. If not specified, Platform Designer uses a default name.

`component_version` *(optional)* The required version of the specified instance type. If not specified, Platform Designer uses the latest version.

**Example**
```
add_component myuart_0 myuart.ip altera_avalon_uart altera_avalon_uart_inst 17.0
```

**Related Information**
- load_component
- load.instantiation
- save_system on page 342
6.7.3.5. duplicate_instance

**Description**
Creates a duplicate instance of the specified instance.

**Usage**
duplicate_instance <instance> [ <name> ]

**Returns**
String  The new instance name.

**Arguments**

*instance*  Specifies the instance name to duplicate.

*name (optional)*  Specifies the name of the duplicate instance.

**Example**
duplicate_instance cpu cpu_0

**Related Information**
- add_instance on page 355
- remove_instance on page 383
6.7.3.6. enable_instance_parameter_update_callback

Description
Enables the update callback for instance parameters.

Usage
enable_instance_parameter_update_callback [<value>]

Returns
No return value.

Arguments

value (optional)  Specifies whether to enable/disable the instance parameters callback. Default option is "1".

Example

enabled_instance_parameter_update_callback

Related Information
- is_instance_parameter_update_callback_enabled on page 382
- set_instance_parameter_value on page 384
6.7.3.7. get_instance_assignment

Description
Returns the assignment value of a child instance. Platform Designer uses assignments to transfer information about hardware to embedded software tools and applications.

Usage
get_instance_assignment <instance> <assignment>

Returns
String  The value of the specified assignment.

Arguments
instance  The instance name.

assignment  The assignment key to query.

Example
get_instance_assignment video_0 embeddedsw.CMacro.colorSpace

Related Information
get_instance_assignments on page 362
6.7.3.8. get_instance_assignments

Description
Returns the list of assignment keys for any defined assignments for the instance.

Usage
get_instance_assignments <instance>

Returns

String[] The list of assignment keys.

Arguments

instance The instance name.

Example
get_instance_assignments sdram

Related Information
get_instance_assignment on page 361
6.7.3.9. get_instance_documentation_links

Description
Returns the list of all documentation links provided by an instance.

Usage
get_instance_documentation_links <instance>

Returns
String[] The list of documentation links.

Arguments
instance The instance name.

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}
6.7.3.10. get_instance_interface_assignment

Description
Returns the assignment value for an interface of a child instance. Platform Designer uses assignments to transfer information about hardware to embedded software tools and applications.

Usage
get_instance_interface_assignment <instance> <interface> <assignment>

Returns
String  The value of the specified assignment.

Arguments
instance  The child instance name.

interface  The interface name.

assignment  The assignment key to query.

Example
get_instance_interface_assignment sram s1 embeddedsw.configuration.isFlash

Related Information
get_instance_interface_assignments on page 365
6.7.3.11. get_instance_interface_assignments

Description
Returns the list of assignment keys for any assignments defined for an interface of a child instance.

Usage
get_instance_interface_assignments \(<instance>\) \(<interface>\)

Returns
String[] The list of assignment keys.

Arguments

instance The child instance name.

interface The interface name.

Example
get_instance_interface_assignments sdram s1

Related Information
get_instance_interface_assignment on page 364
6.7.3.12. get_instance_interface_parameter_property

**Description**
Returns the property value for a parameter in an interface of an instance. Parameter properties are metadata about how Platform Designer uses the parameter.

**Usage**
get_instance_interface_parameter_property <instance> <interface> <parameter> <property>

**Returns**
various  The parameter property value.

**Arguments**

*instance*  The child instance name.

*interface*  The interface name.

*parameter*  The parameter name for the interface.

*property*  The property name for 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 368
- get_instance_interfaces on page 373
- get_parameter_properties on page 424
- Parameter Properties on page 444
6.7.3.13. get_instance_interface_parameter_value

Description
Returns the parameter value of an interface in an instance.

Usage
get_instance_interface_parameter_value <instance> <interface> <parameter>

Returns
various  The parameter value.

Arguments
instance  The child instance name.
interface  The interface name.
parameter  The parameter name for the interface.

Example
get_instance_interface_parameter_value uart_0 s0 setupTime

Related Information
- get_instance_interface_parameters on page 368
- get_instance_interfaces on page 373
6.7.3.14. get_instance_interface_parameters

Description
Returns the list of parameters for an interface in an instance.

Usage
get_instance_interface_parameters <instance> <interface>

Returns
String[] The list of parameter names for parameters in the interface.

Arguments
instance The child instance name.
interface The interface name.

Example
get_instance_interface_parameters uart_0 s0

Related Information
• get_instance_interface_parameter_value on page 367
• get_instance_interfaces on page 373
6.7.3.15. get_instance_interface_port_property

Description
Returns the property value of a port in the interface of a child instance.

Usage
get_instance_interface_port_property <instance> <interface> <port> <property>

Returns
various The port property value.

Arguments
instance The child instance name.
interface The interface name.
port The port name.

property The property name 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 370
- get_port_properties on page 408
- Port Properties on page 448
6.7.3.16. get_instance_interface_ports

**Description**
Returns the list of ports in an interface of an instance.

**Usage**
get_instance_interface_ports <instance> <interface>

**Returns**

String[] The list of port names in the interface.

**Arguments**

*instance* The instance name.

*interface* The interface name.

**Example**

get_instance_interface_ports uart_0 s0

**Related Information**

- get_instance_interface_port_property on page 369
- get_instance_interfaces on page 373
6.7.3.17. get_instance_interface_properties

**Description**
Returns the list of properties that you can query for an interface in an instance.

**Usage**
get_instance_interface_properties

**Returns**

`String[]` The list of property names.

**Arguments**
No arguments.

**Example**

```java
get_instance_interface_properties
```

**Related Information**
- `get_instance_interface_property` on page 372
- `get_instance_interfaces` on page 373
6.7.3.18. get_instance_interface_property

Description
Returns the property value for an interface in a child instance.

Usage
get_instance_interface_property <instance> <interface> <property>

Returns

String The property value.

Arguments

instance The child instance name.

interface The interface name.

property The property name. Refer to Element Properties.

Example

get_instance_interface_property uart_0 s0 DESCRIPTION

Related Information

- get_instance_interface_properties on page 371
- get_instance_interfaces on page 373
- Element Properties on page 439
6. Platform Designer Command-Line Utilities

6.7.3.19. get_instance_interfaces

**Description**

Returns the list of interfaces in an instance.

**Usage**

```plaintext
get_instance_interfaces <instance>
```

**Returns**

`String[]` The list of interface names.

**Arguments**

`instance` The instance name.

**Example**

```plaintext
get_instance_interfaces uart_0
```

**Related Information**

- `get_instance_interface_ports` on page 370
- `get_instance_interface_properties` on page 371
- `get_instance_interface_property` on page 372
6.7.3.20. **get_instance_parameter_property**

**Description**
Returns the property value of a parameter in an instance. Parameter properties are metadata about how Platform Designer uses the parameter.

**Usage**
get_instance_parameter_property `<instance>` `<parameter>` `<property>`

**Returns**
`various` The parameter property value.

**Arguments**

`instance` The instance name.

`parameter` The parameter name.

`property` The property name of the parameter. Refer to Parameter Properties.

**Example**
get_instance_parameter_property uart_0 baudRate ENABLED

**Related Information**
- [get_instance_parameters](#) on page 40
- [get_parameter_properties](#) on page 424
- Parameter Properties on page 444
6.7.3.21. get_instance_parameter_value

**Description**
Returns the parameter value in a child instance.

**Usage**
```plaintext
get_instance_parameter_value <instance> <parameter>
```

**Returns**

*various*  The parameter value.

**Arguments**

*instance*  The instance name.

*parameter*  The parameter name.

**Example**

```plaintext
get_instance_parameter_value pixel_converter input_DPI
```

**Related Information**

- get_instance_parameters on page 40
- set_instance_parameter_value on page 384
6.7.3.22. get_instance_parameter_values

**Description**
Returns a list of the parameters' values in a child instance.

**Usage**
get_instance_parameter_values <instance> <parameters>

**Returns**

`String[]` A list of the parameters' value.

**Arguments**

*instance*  The child instance name.

*parameter*  A list of parameter names in the instance.

**Example**

```
get_instance_parameter_value uart_0 [list param1 param2]
```

**Related Information**

- `get_instance_parameters` on page 40
- `set_instance_parameter_value` on page 384
- `set_instance_parameter_values` on page 385
6.7.3.23. get_instance_parameters

**Description**
Returns the names of all parameters for a child instance that the parent can manipulate. This command omits derived parameters and parameters that have the SYSTEM_INFO parameter property set.

**Usage**
get_instance_parameters <instance>

**Returns**

*instance*  The list of parameters in the instance.

**Arguments**

*instance*  The instance name.

**Example**

get_instance_parameters uart_0

**Related Information**
- get_instance_parameter_property on page 374
- get_instance_parameter_value on page 39
- set_instance_parameter_value on page 384
6.7.3.24. get_instance_port_property

**Description**
Returns the property value of a port contained by an interface in a child instance.

**Usage**
get_instance_port_property <instance> <port> <property>

**Returns**
various  The property value for the port.

**Arguments**

instance  The child instance name.

port  The port name.

property  The property name. Refer to Port Properties.

**Example**
get_instance_port_property uart_0 tx WIDTH

**Related Information**
- get_instance_interface_ports on page 370
- get_port_properties on page 408
- Port Properties on page 448
6.7.3.25. get_instance_properties

**Description**
Returns the list of properties for a child instance.

**Usage**
get_instance_properties

**Returns**

String[]  The list of property names for the child instance.

**Arguments**
No arguments.

**Example**
get_instance_properties

**Related Information**
get_instance_property on page 380
6.7.3.26. **get_instance_property**

**Description**
Returns the property value for a child instance.

**Usage**
get_instance_property `<instance>` `<property>`

**Returns**

*String*  The property value.

**Arguments**

*instance*  The child instance name.

*property*  The property name. Refer to *Element Properties*.

**Example**

```
get_instance_property uart_0 ENABLED
```

**Related Information**

-  *get_instance_properties* on page 379
-  *Element Properties* on page 439
6.7.3.27. get_instances

Description
Returns the list of the instance names for all the instances in the system.

Usage
get_instances

Returns
String[] The list of child instance names.

Arguments
No arguments.

Example
get_instances

Related Information
• add_instance on page 355
• remove_instance on page 383
6.7.3.28. is_instance_parameter_update_callback_enabled

Description
Returns true if you enable the update callback for instance parameters.

Usage
is_instance_parameter_update_callback_enabled

Returns
boolean 1 if you enable the callback; 0 if you disable the callback.

Arguments
No arguments

Example
is_instance_parameter_update_callback_enabled

Related Information
enable_instance_parameter_update_callback on page 360
6.7.3.29. remove_instance

**Description**
Removes an instance from the system.

**Usage**
remove_instance <instance>

**Returns**
No return value.

**Arguments**

*instance*   The child instance name to remove.

**Example**

```
remove_instance cpu
```

**Related Information**
- [add_instance](#) on page 355
- [get_instances](#) on page 381
6.7.3.30. set_instance_parameter_value

**Description**
Sets the parameter value 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 child instance name.

*parameter*  The parameter name.

*value*  The parameter value.

**Example**

```
set_instance_parameter_value uart_0 baudRate 9600
```

**Related Information**

- [get_instance_parameter_value](#) on page 39
- [get_instance_parameter_property](#) on page 374
6.7.3.31. set_instance_parameter_values

Description
Sets a list of parameter values 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_pairs>

Returns
No return value.

Arguments

instance  The child instance name.

parameter_value_pairs  The pairs of parameter name and value to set.

Example
set_instance_parameter_value uart_0 [list baudRate 9600 parity odd]

Related Information
- get_instance_parameter_value on page 39
- get_instance_parameter_values on page 376
- get_instance_parameters on page 40
6.7.3.32. set_instance_property

**Description**

Sets the property value 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 Platform Designer interconnect.

**Usage**

```
set_instance_property <instance> <property> <value>
```

**Returns**

No return value.

**Arguments**

- **instance**  The child instance name.
- **property**  The property name. Refer to *Instance Properties*.
- **value**  The property value.

**Example**

```
set_instance_property cpu ENABLED false
```

**Related Information**

- [get_instance_parameters](#) on page 40
- [get_instance_property](#) on page 380
- [Instance Properties](#) on page 440
6.7.4. Connections

This section lists the commands that allow you to manipulate the interface connections in your Platform Designer system.

- add_connection on page 388
- auto_connect on page 389
- get_connection_parameter_property on page 390
- get_connection_parameter_value on page 391
- get_connection_parameters on page 392
- get_connection_properties on page 393
- get_connection_property on page 394
- get_connections on page 395
- remove_connection on page 396
- remove_dangling_connections on page 397
- set_connection_parameter_value on page 398
6.7.4.1. add_connection

**Description**
Connects the named interfaces using an appropriate connection type. Both interface names consist of an instance name, followed by the interface name that the module provides.

**Usage**
add_connection `<start>` `[<end>]`

**Returns**
No return value.

**Arguments**

*start* The start interface that you connect, in `<instance_name>.<interface_name>` format. If you do not specify the end argument, the connection must be of the form `<instance1>.<interface> / <instance2>.<interface>`.

*end (optional)* The end interface that you connect, in `<instance_name>.<interface_name>` format.

**Example**
add_connection dma.read_master sdram.s1

**Related Information**
- get_connection_parameter_value on page 391
- get_connection_property on page 394
- get_connections on page 395
- remove_connection on page 396
- set_connection_parameter_value on page 398
6.7.4.2. auto_connect

**Description**
Creates connections from an instance or instance interface to matching interfaces of other instances in the system. For example, Avalon-MM slaves connect to Avalon-MM masters.

**Usage**
```plaintext
auto_connect <element>
```

**Returns**
No return value.

**Arguments**

*element* The instance interface name, or the instance name.

**Example**
```plaintext
auto_connect sdram
auto_connect uart_0.s1
```

**Related Information**
add_connection on page 388
6.7.4.3. get_connection_parameter_property

**Description**
Returns the property value of a parameter in a connection. Parameter properties are metadata about how Platform Designer uses the parameter.

**Usage**
get_connection_parameter_property <connection> <parameter> <property>

**Returns**
various  The parameter property value.

**Arguments**

*connection*  The connection to query.

*parameter*  The parameter name.

*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 391
- [get_connection_property](#) on page 394
- [get_connections](#) on page 395
- [get_parameter_properties](#) on page 424
- Parameter Properties on page 444
6.7.4.4. `get_connection_parameter_value`

**Description**
Returns the parameter value of the connection. Parameters represent aspects of the connection that you can modify, such as the base address for an Avalon-MM connection.

**Usage**

```
get_connection_parameter_value <connection> <parameter>
```

**Returns**
`various` The parameter value.

**Arguments**

- `connection` The connection to query.
- `parameter` The parameter name.

**Example**

```
get_connection_parameter_value cpu.data_master/dma0.csr baseAddress
```

**Related Information**
- `get_connection_parameters` on page 392
- `get_connections` on page 395
- `set_connection_parameter_value` on page 398
6.7.4.5. get_connection_parameters

**Description**
Returns the list of parameters of a connection.

**Usage**
get_connection_parameters `<connection>`

**Returns**

`String[]` The 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 390
- `get_connection_parameter_value` on page 391
- `get_connection_property` on page 394
6.7.4.6. get_connection_properties

**Description**
Returns the properties list of a connection.

**Usage**
get_connection_properties

**Returns**

`String[]` The list of connection properties.

**Arguments**
No arguments.

**Example**

```
get_connection_properties
```

**Related Information**
- `get_connection_property` on page 394
- `get_connections` on page 395
6.7.4.7. get_connection_property

**Description**
Returns the property value of a connection. Properties represent aspects of the connection that you can modify, such as the connection type.

**Usage**
get_connection_property <connection> <property>

**Returns**
*String*  The connection property value.

**Arguments**
*connection*  The connection to query.

*property*  The connection property name. Refer to *Connection Properties*.

**Example**
```
get_connection_property cpu.data_master/dma0.csr TYPE
```

**Related Information**
- [get_connection_properties on page 393](#)
- [Connection Properties on page 436](#)
6.7.4.8. get_connections

Description
Returns the list of all connections in the system if you do not specify any element. If you specify a child instance, for example `cpu`, Platform Designer returns all connections to any interface on the instance. If you specify an interface of a child instance, for example `cpu.instruction_master`, Platform Designer returns all connections to that interface.

Usage
get_connections [\textit{<element>}] 

Returns
\textit{String[]}  The list of connections.

Arguments
\textit{element (optional)}  The child instance name, or the qualified interface name on a child instance.

Example
get_connections
get_connections cpu
get_connections cpu.instruction_master

Related Information
- \textit{add_connection} on page 388
- \textit{remove_connection} on page 396
6.7.4.9. remove_connection

Description
Removes a connection from the system.

Usage
remove_connection <connection>

Returns
No return value.

Arguments

connection The connection name to remove.

Example
remove_connection cpu.data_master/sdram.s0

Related Information
• add_connection on page 388
• get_connections on page 395
6.7.4.10. 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

Related Information
- add_connection on page 388
- get_connections on page 395
- remove_connection on page 396
6.7.4.11. set_connection_parameter_value

**Description**
Sets the parameter value for a connection.

**Usage**

```
set_connection_parameter_value <connection> <parameter> <value>
```

**Returns**
No return value.

**Arguments**

- `connection`  The connection name.
- `parameter`  The parameter name.
- `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 391
- [get_connection_parameters](#) on page 392
6.7.5. Top-level Exports

This section lists the commands that allow you to manipulate the exported interfaces in your Platform Designer system.

add_interface on page 400
get_exported_interface_sysinfo_parameter_value on page 401
get_exported_interface_sysinfo_parameters on page 402
get_interface_port_property on page 403
get_interface_ports on page 404
get_interface_properties on page 405
get_interface_property on page 406
get_interfaces on page 407
get_port_properties on page 408
remove_interface on page 409
set_interface_port_property on page 410
set_interface_property on page 411
6.7.5.1. add_interface

Description
Adds an interface to your system, which Platform Designer 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 Platform Designer 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 404
- `get_interface_properties` on page 405
- `get_interface_property` on page 406
- `set_interface_property` on page 411
6.7.5.2. get_exported_interface_sysinfo_parameter_value

Description
Gets the value of a system info parameter for an exported interface.

Usage
get_exported_interface_sysinfo_parameter_value <interface>
<parameter>

Returns
various The system info parameter value.

Arguments
interface Specifies the name of the exported interface.

parameter Specifies the name of the system info parameter. Refer to System Info Type.

Example
get_exported_interface_sysinfo_parameter_value clk clock_rate

Related Information
- get_exported_interface_sysinfo_parameters on page 402
- set_exported_interface_sysinfo_parameter_value
- System Info Type Properties on page 450
6.7.5.3. get_exported_interface_sysinfo_parameters

Description
Returns the list of system info parameters for an exported interface.

Usage
get_exported_interface_sysinfo_parameters <interface> [ <type> ]

Returns
String[]  The list of system info parameter names.

Arguments
interface  Specifies the name of the exported interface.

type (optional)  Specifies the parameters type to return. If you do not specify this option, the command returns all the parameters. Refer to Access Type.

Example
get_exported_interface_sysinfo_parameters clk

Related Information
• get_exported_interface_sysinfo_parameter_value on page 401
• set_exported_interface_sysinfo_parameter_value
• Access Type on page 456
6.7.5.4. 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
various The property value.

Arguments
interface The name of a top-level interface of the system.

port The port name in the interface.

property The property name on the port. Refer to Port Properties.

Example
get_interface_port_property uart_exports tx DIRECTION

Related Information
- get_interface_ports on page 404
- get_port_properties on page 408
- Port Properties on page 448
6.7.5.5. get_interface_ports

**Description**
Returns the names of all the added ports to a given interface.

**Usage**
get_interface_ports <interface>

**Returns**

String[] The list of port names.

**Arguments**

interface The top-level interface name of the system.

**Example**

get_interface_ports export_clk_out

**Related Information**

- get_interface_port_property on page 403
- get_interfaces on page 407
6.7.5.6. get_interface_properties

Description
Returns the names of all the available interface properties common to all interface types.

Usage
get_interface_properties

Returns

`String[]` The list of interface properties.

Arguments
No arguments.

Example

```
get_interface_properties
```

Related Information
- `get_interface_property` on page 406
- `set_interface_property` on page 411
6.7.5.7. get_interface_property

**Description**

Returns the value of a single interface property from the specified interface.

**Usage**

```
get_interface_property <interface> <property>
```

**Returns**

`various` The property value.

**Arguments**

- `interface` The name of a top-level interface of 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 405
- `set_interface_property` on page 411
- Interface Properties on page 441
6.7.5.8. get_interfaces

**Description**
Returns the list of top-level interfaces in the system.

**Usage**
get_interfaces

**Returns**

String[] The list of the top-level interfaces exported from the system.

**Arguments**
No arguments.

**Example**
get_interfaces

**Related Information**
- add_interface on page 400
- get_interface_ports on page 404
- get_interface_property on page 406
- remove_interface on page 409
- set_interface_property on page 411
6.7.5.9. get_port_properties

Description
Returns the list of properties that you can query for ports.

Usage
get_port_properties

Returns

String[] The list of port properties.

Arguments
No arguments.

Example

get_port_properties

Related Information
- get_instance_interface_port_property on page 369
- get_instance_interface_ports on page 370
- get_instance_port_property on page 378
- get_interface_port_property on page 403
- get_interface_ports on page 404
6.7.5.10. removeinterface

**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 400
- [get_interfaces](#) on page 407
## 6.7.5.11. set_interface_port_property

### Description
Sets the port property in a top-level interface of the system.

### Usage
```
set_interface_port_property <interface> <port> <property> <value>
```

### Returns
No return value

### Arguments
- **interface**  Specifies the top-level interface name of the system.
- **port**  Specifies the port name in a top-level interface of the system.
- **property**  Specifies the property name of the port. Refer to Port Properties.
- **value**  Specifies the property value.

### Example
```
set_interface_port_property clk clk_clk NAME my_clk
```

### Related Information
- Port Properties on page 461
- get_interface_ports on page 404
- get_interfaces on page 407
- get_port_properties on page 408
6.7.5.12. 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 property value.

Example

set_interface_property clk_out EXPORT_OF clk.clk_out

Related Information

- add_interface on page 400
- get_interface_properties on page 405
- get_interface_property on page 406
- Interface Properties on page 441
6.7.6. Validation

This section lists the commands that allow you to validate the components, instances, interfaces and connections in a Platform Designer system.

set_validation_property on page 413
validate_connection on page 414
validate_instance on page 415
validate_instance_interface on page 416
validate_system on page 417
6.7.6.1. 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 417
• Validation Properties on page 453
6.7.6.2. validate_connection

**Description**
Validates the specified connection and returns validation messages.

**Usage**
validate_connection <connection>

**Returns**
A list of validation messages.

**Arguments**

connection  The connection name to validate.

**Example**
validate_connection cpu.data_master/sdram.s1

**Related Information**
- validate_instance on page 415
- validate_instance_interface on page 416
- validate_system on page 417
6.7.6.3. validate_instance

**Description**
Validates the specified child instance and returns validation messages.

**Usage**
validate_instance <instance>

**Returns**
A list of validation messages.

**Arguments**

*instance*  The child instance name to validate.

**Example**

validate_instance cpu

**Related Information**

- [validate_connection on page 414](#)
- [validate_instance_interface on page 416](#)
- [validate_system on page 417](#)
6.7.6.4. validate_instance_interface

**Description**
Validates an interface of an instance and returns validation messages.

**Usage**
validate_instance_interface <instance> <interface>

**Returns**
A list of validation messages.

**Arguments**

*instance* The child instance name.

*interface* The interface to validate.

**Example**

```
validate_instance_interface cpu data_master
```

**Related Information**
- validate_connection on page 414
- validate_instance on page 415
- validate_system on page 417
6.7.6.5. validate_system

Description
Validates the system and returns validation messages.

Usage
validate_system

Returns
A list of validation messages.

Arguments
No arguments.

Example
validate_system

Related Information
- validate_connection on page 414
- validate_instance on page 415
- validate_instance_interface on page 416
6.7.7. Miscellaneous

This section lists the miscellaneous commands that you can use for your Platform Designer systems.

- auto_assign_base_addresses on page 419
- auto_assign_irqs on page 420
- auto_assign_system_base_addresses on page 421
- get_interconnect_requirement on page 422
- get_interconnect_requirements on page 423
- get_parameter_properties on page 424
- lock_avalon_base_address on page 425
- send_message on page 43
- set_interconnect_requirement on page 427
- set_use_testbench_naming_pattern on page 428
- unlock_avalon_base_address on page 429
- get_testbench_dutname on page 430
- get_use_testbench_naming_pattern on page 431
6.7.7.1. auto_assign_base_addresses

**Description**
Assigns base addresses to all memory-mapped interfaces of 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 421
- [lock_avalon_base_address](#) on page 425
- [unlock_avalon_base_address](#) on page 429
6.7.7.2. auto_assign_irqs

**Description**
Assigns interrupt numbers to all connected interrupt senders of 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
6.7.7.3. auto_assign_system_base_addresses

**Description**
Assigns legal base addresses to all memory-mapped interfaces of 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 419
- `lock_avalon_base_address` on page 425
- `unlock_avalon_base_address` on page 429
6.7.7.4. get_interconnect_requirement

**Description**
Returns the value of an interconnect requirement for a system or interface of a child instance.

**Usage**
get_interconnect_requirement <element_id> <requirement>

**Returns**
String 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**
```bash
get_interconnect_requirement {$system} qsys_mm.maxAdditionalLatency
```
6.7.7.5. get_interconnect_requirements

Description
Returns the list of all interconnect requirements in the system.

Usage
get_interconnect_requirements

Returns
String[] A flattened list of interconnect requirements. Every sequence of three elements in the list corresponds to one interconnect requirement. The first element in the sequence is the element identifier. The second element is the requirement name. The third element is the value. You can loop over the returned list with a foreach loop, for example:

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

Arguments
No arguments.

Example
get_interconnect_requirements
6.7.7.6. get_parameter_properties

**Description**
Returns the list of properties that you can query for any parameters, for example parameters of instances, interfaces, instance interfaces, and connections.

**Usage**
get_parameter_properties

**Returns**

*String[]* The list of parameter properties.

**Arguments**
No arguments.

**Example**

get_parameter_properties

**Related Information**
- [get_connection_parameter_property on page 390](#)
- [get_instance_interface_parameter_property on page 366](#)
- [get_instance_parameter_property on page 374](#)
6.7.7.7. lock_avalon_base_address

Description
Prevents the memory-mapped base address from being changed for connections to the specified interface of an instance when Platform Designer 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 419
• auto_assign_system_base_addresses on page 421
• unlock_avalon_base_address on page 429
6.7.7.8. send_message

Description
Sends a message to the user of the component. The message text is normally HTML. You can use the `<b>` element to provide emphasis. If you do not want the message text to be HTML, then pass a list like `{Info Text}` as the message level.

Usage
`send_message <level> <message>`

Returns
No return value.

Arguments

`level` Intel Quartus Prime supports the following message levels:
- 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!"
```
6.7.7.9. set_interconnect_requirement

**Description**
Sets the value of an interconnect requirement for a system or an interface of 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 requirement value.

**Example**

```
set_interconnect_requirement {$system} qsys_mm.clockCrossingAdapter HANDSHAKE
```
6.7.7.10. 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 command, 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.
6.7.11. unlock_avalon_base_address

**Description**
Allows the memory-mapped base address to change for connections to the specified interface of an instance when Platform Designer 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 419
- `auto_assign_system_base_addresses` on page 421
- `lock_avalon_base_address` on page 425
6.7.7.12. get_testbench_dutname

Description
Returns the currently set dutname for the test-bench systems. Use this command only when creating test-bench systems.

Usage
get_testbench_dutname

Returns
String  The currently set dutname. Returns NULL if empty.

Arguments
No arguments.

Example
get_testbench_dutname

Related Information
• get_use_testbench_naming_pattern on page 431
• set_use_testbench_naming_pattern on page 428
6.7.7.13. get_use_testbench_naming_pattern

**Description**
Verifies if the test-bench naming pattern is set to be used. Use this command only when creating test-bench systems.

**Usage**
get_use_testbench_naming_pattern

**Returns**

*boolean*  True, if the test-bench naming pattern is set to be used.

**Arguments**
No arguments.

**Example**
get_use_testbench_naming_pattern

**Related Information**
- get_testbench_dutname on page 430
- set_use_testbench_naming_pattern on page 428

6.7.8. Wire-Level Connection Commands

Wire-level commands accept optional input ports and wire-level expressions as arguments for the qsys-script utility and in _hw.tcl files.

You can use wire-level commands to:

- Apply a wire-level expression to a port with set_wirelevel_expression.
- Retrieve a list of expressions from a port, instance, or all expressions in the current level of system hierarchy with get_wirelevel_expression.
- Remove a list of expressions from a port, instance, or all expressions in the current level of system hierarchy with remove_wirelevel_expression.

**Note:**
The following restrictions apply when using wire-level commands _hw.tcl files:

- Wire-level commands are only valid in a composition callback.
- Wire-level expressions can only be applied to instances created by add_instance.

**Related Information**
Create a Composed Component or Subsystem on page 312

6.7.8.1. set_wirelevel_expression

**Description**
Applies a wire-level expression to an optional input port or instance in the system.
### Usage

set_wirelevel_expression <instance_or_port_bitselection> <expression>

### Returns

No return value.

### Arguments

- **instance_or_port_bitselection**
  Specify the instance or port to which the wire-level expression using the <instance_name>.<port_name>[<bit_selection>] format. The bit selection can be a bit-select, for example [0], or a partial range defined in descending order, for example [7:0]. If no bit selection is specified, the full range of the port is selected.

- **expression**
  The expression to be applied to an optional input port.

### Examples

```
set_wirelevel_expression {module0.portA[7:0]} "8'b0"
set_wirelevel_expression module0.portA "8'b0"
set_wirelevel_expression {module0.portA[0]} "1'b0"
```

### 6.7.8.2. get_wirelevel_expressions

#### Description

Retrieve a list of wire-level expressions from an optional input port, instance, or all expressions in the current level of system hierarchy. If the port bit selection is specified as an argument, the range must be identical to what was used in the set_wirelevel_expression statement.

#### Usage

get_wirelevel_expressions <instance_or_port_bitselection>

#### Returns

**String[]** A flattened list of wire-level expressions. Every item in the list consists of right- and left-hand clauses of a wire-level expression. You can loop over the returned list using foreach(port expr) $return_list{}.

#### Arguments

- **instance_or_port_bitselection**
  Specifies which instance or port from which a list of wire-level expressions are retrieved using the <instance_name>.<port_name>[<bit_selection>] format.
• If no `<port_name>[<bit_selection>]` is specified, the command causes the return of all expressions from the specified instance.

• If no argument is present, the command causes the return of all expressions from the current level of system hierarchy.

The bit selection can be a bit-select, for example `[0]`, or a partial range defined in descending order, for example `[7:0]`. If no bit selection is specified, the full range of the port is selected.

**Example**
```
get_wirelevel_expressions
get_wirelevel_expressions module0
get_wirelevel_expressions {module0.portA[7:0]}
```

### 6.7.8.3. remove_wirelevel_expressions

**Description**
Remove a list of wire-level expressions from an optional input port, instance, or all expressions in the current level of system hierarchy. If the port bit selection is specified as an argument, the range must be identical to what was used in the `set_wirelevel_expressions` statement.

**Usage**
```
remove_wirelevel_expressions <instance_or_port_bitselection>
```

**Returns**
No return value.

**Arguments**

*instance_or_port_bitselection* Specifies which instance or port from which a list of wire-level expressions are removed using the `<instance_name>.<port_name>[<bit_selection>]` format.

• If no `<port_name>[<bit_selection>]` is specified, the command causes the removal of all expressions from the specified instance.

• If no argument is present, the command causes the return of all expressions from the current level of system hierarchy.

The bit selection can be a bit-select, for example `[0]`, or a partial range defined in descending order, for example `[7:0]`. If no bit selection is specified, the full range of the port is selected.
Examples

```plaintext
remove_wirelevel_expressions
remove_wirelevel_expressions module0
remove_wirelevel_expressions {module0.portA[7:0]}
```
6.8. Platform Designer Scripting Property Reference

Interface properties work differently for _hw.tcl scripting than with Platform Designer scripting. In _hw.tcl, interfaces do not distinguish between properties and parameters. In Platform Designer scripting, the properties and parameters are unique.

The following are the Platform Designer scripting properties:

- Connection Properties on page 436
- Design Environment Type Properties on page 437
- Direction Properties on page 438
- Element Properties on page 439
- Instance Properties on page 440
- Interface Properties on page 441
- Message Levels Properties on page 442
- Module Properties on page 443
- Parameter Properties on page 444
- Parameter Status Properties on page 446
- Parameter Type Properties on page 447
- Port Properties on page 448
- Project Properties on page 449
- System Info Type Properties on page 450
- Units Properties on page 452
- Validation Properties on page 453
- Interface Direction on page 454
- File Set Kind on page 455
- Access Type on page 456
- Instantiation HDL File Properties on page 457
- Instantiation Interface Duplicate Type on page 458
- Instantiation Interface Properties on page 459
- Instantiation Properties on page 460
- Port Properties on page 461
- VHDL Type on page 462
### 6.8.1. 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>Indicates the end interface of the connection.</td>
</tr>
<tr>
<td>string</td>
<td>NAME</td>
<td>Indicates the name of the connection.</td>
</tr>
<tr>
<td>string</td>
<td>START</td>
<td>Indicates 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>
6.8.2. Design Environment Type Properties

**Description**

IP cores use the design environment to identify the most appropriate interfaces to connect to the parent system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE</td>
<td>Supports native IP interfaces.</td>
</tr>
<tr>
<td>QSYS</td>
<td>Supports standard Platform Designer interfaces.</td>
</tr>
</tbody>
</table>
### 6.8.3. Direction Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDIR</td>
<td>Indicates the direction for a bidirectional signal.</td>
</tr>
<tr>
<td>INOUT</td>
<td>Indicates the direction for an input signal.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Indicates the direction for an output signal.</td>
</tr>
</tbody>
</table>
6.8.4. 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 Platform Designer Component Editor.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>Indicates whether the instance is enabled.</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, 16.1.</td>
</tr>
</tbody>
</table>
## 6.8.5. 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 Platform Designer automatically exports the unconnected interfaces on the instance.</td>
</tr>
<tr>
<td>Boolean</td>
<td>ENABLED</td>
<td>If true, Platform Designer includes this instance in the generated system.</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
<td>The name of the system, which Platform Designer uses as the name of the top-level module in the generated HDL.</td>
</tr>
</tbody>
</table>
### 6.8.6. Interface Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 <code>add_interface</code> command. You must use the format: <code>&lt;instanceName.interfaceName&gt;</code>. For example: <code>set_interface_property CSC_input EXPORT_OF my_colorSpaceConverter.input_port</code></td>
</tr>
</tbody>
</table>
### 6.8.7. 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 enabled.</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>
</tbody>
</table>
## 6.8.8. Module Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>GENERATION_ID</td>
<td>The generation ID for the system.</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
<td>The name of the instance.</td>
</tr>
</tbody>
</table>
## 6.8.9. 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 <code>isNonVolatileStorage</code>. 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, Platform Designer 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. Setting this property to false 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, <code>11:15</code>. This property also specifies the legal values and description strings for integers, for example, <code>{0:None 1:Monophonic 2:Stereo 4:Quadrophonic}</code>, where 0, 1, 2, and 4 are the legal values. You can assign description strings in the parameter editor for string variables. For example, <code>ALLOWED_RANGES = {'dev1:Cyclone IV GX**dev2:Stratix® V GT'}</code></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></td>
<td></td>
<td>• boolean—for integer parameters whose value are 0 or 1. The parameter displays as an option that you can turn on or off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• radio—displays a parameter with a list of values as radio buttons.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• hexadecimal—for integer parameters, displays and interprets the value as a hexadecimal number, for example: <code>0x00000010</code> instead of 16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• fixed_size—for string_list and integer_list parameters, the fixed_size DISPLAY_HINT eliminates the <code>Add</code> and <code>Remove</code> buttons from tables.</td>
</tr>
<tr>
<td>String</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 disabled. The parameter 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>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Boolean</td>
<td>HDL_PARAMETER</td>
<td>When True, Platform Designer 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. Older 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></td>
<td></td>
<td>SYSTEM_INFO &lt;info-type&gt;</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>String</td>
<td>WIDTH</td>
<td>Indicates the width of the logic vector for the STD_LOGIC_VECTOR parameter.</td>
</tr>
</tbody>
</table>

**Related Information**

- [System Info Type Properties](#) on page 450
- [Parameter Type Properties](#) on page 447
- [Units Properties](#) on page 452
### 6.8.10. 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>
## 6.8.11. 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 true or false.</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 $2^{31} - 1$ ($0x7fffffff$).</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>A 32-bit number that contains values 1 to $2^{31} - 1$ ($0x7fffffff$).</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 WIDTH 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>
## 6.8.12. 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 Direction Properties.</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>

**Related Information**

Direction Properties on page 438
### 6.8.13. 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 Intel Quartus Prime project that contains the Platform Designer system.</td>
</tr>
<tr>
<td>String</td>
<td>DEVICE_FAMILY</td>
<td>The device family name in the Intel Quartus Prime project that contains the Platform Designer system.</td>
</tr>
</tbody>
</table>
### 6.8.14. 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 SYSTEM_INFO parameter property.</td>
</tr>
<tr>
<td>Integer</td>
<td>ADDRESS_WIDTH</td>
<td>The number of address bits that Platform Designer 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 Platform Designer interconnect. Refer to Avalon Interface Specifications.</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 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 SYSTEM_INFO 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 Platform Designer interconnect that conforms to Avalon Interface Specifications.</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 Design Environment Type Properties.</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 array 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>An integer that stores a hash of the generation time that Platform Designer uses as a unique ID for a generation run.</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 representing the reset domain for the interface specified in the SYSTEM_INFO parameter property if this instance has interfaces on multiple reset domains.</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>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 SYSTEM_INFO 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 437
- [Avalon Interface Specifications](#)
- [Platform Designer Interconnect](#) on page 124
# 6.8.15. 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 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>
6.8.16. 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 true, Platform Designer runs system validation and elaboration after each scripting command. When false, Platform Designer runs system validation with validation scripting commands. Some queries affected by system elaboration may be incorrect if automatic validation is disabled. You can disable validation to make a system script run faster.</td>
</tr>
</tbody>
</table>
### 6.8.17. Interface Direction

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>INPUT</td>
<td>Indicates that the interface is a slave (input, transmitter, sink, or end).</td>
</tr>
<tr>
<td>String</td>
<td>OUTPUT</td>
<td>Indicates that the interface is a master (output, receiver, source, or start).</td>
</tr>
</tbody>
</table>
### 6.8.18. File Set Kind

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE_DESIGN</td>
<td>This file-set contains example design files.</td>
</tr>
<tr>
<td>QUARTUS_SYNTH</td>
<td>This file-set contains files that Platform Designer uses for Intel Quartus Prime Synthesis</td>
</tr>
<tr>
<td>SIM_VERILOG</td>
<td>This file-set contains files that Platform Designer uses for Verilog HDL Simulation.</td>
</tr>
<tr>
<td>SIM_VHDL</td>
<td>This file-set contains files that Platform Designer uses for VHDL Simulation.</td>
</tr>
</tbody>
</table>
### 6.8.19. Access Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>READ_ONLY</td>
<td>Indicates that the parameter can be only read-only.</td>
</tr>
<tr>
<td>String</td>
<td>WRITABLE</td>
<td>Indicates that the parameter has read/write properties.</td>
</tr>
</tbody>
</table>
### 6.8.20. Instantiation HDL File Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTAINS_INLINE_CONFIGURATION</td>
<td>Boolean</td>
<td>Returns True if the HDL file contains inline configuration.</td>
</tr>
<tr>
<td>IS_CONFIGURATION_PACKAGE</td>
<td>Boolean</td>
<td>Returns True if the HDL file is a configuration package.</td>
</tr>
<tr>
<td>IS_TOP_LEVEL</td>
<td>Boolean</td>
<td>Returns True if the HDL file is the top-level HDL file.</td>
</tr>
<tr>
<td>OUTPUT_PATH</td>
<td>String</td>
<td>Specifies the output path of the HDL file.</td>
</tr>
<tr>
<td>TYPE</td>
<td>String</td>
<td>Specifies the HDL file type of the HDL file.</td>
</tr>
</tbody>
</table>
### 6.8.21. Instantiation Interface Duplicate Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>CLONE</td>
<td>Creates a copy of an interface and all the interface ports.</td>
</tr>
<tr>
<td>String</td>
<td>MIRROR</td>
<td>Creates a copy of an interface with all the port roles and directions reversed.</td>
</tr>
</tbody>
</table>
### 6.8.22. Instantiation Interface Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>DIRECTION</td>
<td>The direction of the interface.</td>
</tr>
<tr>
<td>String</td>
<td>TYPE</td>
<td>The type of the interface.</td>
</tr>
</tbody>
</table>
### 6.8.23. Instantiation Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HDL_COMPILATION_LIBRARY</td>
<td>Indicates the HDL compilation library name of the generic component.</td>
</tr>
<tr>
<td>String</td>
<td>HDL_ENTITY_NAME</td>
<td>Indicates the HDL entity name of the Generic Component.</td>
</tr>
<tr>
<td>String</td>
<td>IP_FILE</td>
<td>Indicates the .ip file path that implements the generic component.</td>
</tr>
</tbody>
</table>
### 6.8.24. Port Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>DIRECTION</td>
<td>Specifies the direction of the signal</td>
</tr>
<tr>
<td>String</td>
<td>NAME</td>
<td>Renames a top-level port. Only use with <code>set_interface_port_property</code></td>
</tr>
<tr>
<td>String</td>
<td>ROLE</td>
<td>Specifies the type of the signal. Each interface type defines a set of interface types for its ports.</td>
</tr>
<tr>
<td>String</td>
<td>VHDL_TYPE</td>
<td>Specifies the VHDL type of the signal. Can be either <code>STANDARD_LOGIC</code>, or <code>STANDARD_LOGIC_VECTOR</code>.</td>
</tr>
<tr>
<td>Integer</td>
<td>WIDTH</td>
<td>Specifies the width of the signal in bits.</td>
</tr>
</tbody>
</table>

**Related Information**

Direction Properties on page 438
6.8.25. VHDL Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD_LOGIC</td>
<td>Represents the value of a digital signal in a wire.</td>
</tr>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>Represents an array of digital signals and variables.</td>
</tr>
</tbody>
</table>

6.9. Platform Designer Command-Line Interface Revision History

The following revision history applies to this chapter:

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018.12.15</td>
<td>18.1.0</td>
<td>First release as separate chapter.</td>
</tr>
<tr>
<td>2016.05.03</td>
<td>16.0.0</td>
<td>• Qsys Command-Line Utilities updated with latest supported command-line options.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>• Added command-line utilities, and scripts.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Initial release of content.</td>
</tr>
</tbody>
</table>
7. Component Interface Tcl Reference

Tcl commands allow you to perform a wide range of functions in Platform Designer. Command descriptions contain the Platform Designer phases where you can use the command, for example, main program, elaboration, composition, or fileset callback. This reference denotes optional command arguments in brackets [ ].

Note: Intel now refers to Qsys as Platform Designer (Standard).

Platform Designer supports Avalon, AMBA 3 AXI (version 1.0), AMBA 4 AXI (version 2.0), AMBA 4 AXI-Lite (version 2.0), AMBA 4 AXI-Stream (version 1.0), and AMBA 3 APB (version 1.0) interface specifications.

For more information about procedures for creating IP component _hw.tcl files in the Platform Designer Component Editor, and supported interface standards, refer to Creating Platform Designer Components and Platform Designer Interconnect.

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 Platform Designer Components on page 282
- Platform Designer Interconnect on page 124
- Publishing Component Information to Embedded Software in Nios II Gen2 Software Developer's Handbook

7.1. Platform Designer _hw.tcl Command Reference
7.1.1. Interfaces and Ports

- `add_interface` on page 465
- `add_interface_port` on page 467
- `get_interfaces` on page 469
- `get_interface_assignment` on page 470
- `get_interface_assignments` on page 471
- `get_interface_ports` on page 472
- `get_interface_properties` on page 473
- `get_interface_property` on page 474
- `get_port_properties` on page 475
- `get_port_property` on page 476
- `set_interface_assignment` on page 477
- `set_interface_property` on page 479
- `set_port_property` on page 480
- `set_interface_upgrade_map` on page 481

**Related Information**

Interface Properties on page 561
7.1.1.1. 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  (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 Platform Designer-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 467
- `get_interface_assignments` on page 471
- `get_interface_properties` on page 473
- `get_interfaces` on page 469
7. Component Interface Tcl Reference

7.1.1.2. add_interface_port

Description
Adds a port to an interface on your module. The name must match the name of a signal on the top-level module in the HDL of your IP component. The port width and direction must be set before the end of the elaboration phase. You can set the port width as follows:

- In the Main program, you can set the port width to a fixed value or a width expression.
- If the port width is set to a fixed value in the Main program, you can update the width in the elaboration callback.

Availability
Main Program, Elaboration

Usage
add_interface_port <interface> <port> [<<signal_type> <direction> <width_expression>]

Returns
Arguments

interface  The name of the interface to which this port belongs.

port  The name of the port. This name must match a signal in your top-level HDL for this IP component.

signal_type  (optional)  The type of signal for this port, which must be unique. Refer to the Avalon Interface Specifications for the signal types available for each interface type.

direction  (optional)  The direction of the signal. Refer to Direction Properties.

width_expression  (optional)  The width of the port, in bits. The width may be a fixed value, or a simple arithmetic expression of parameter values.

Example

fixed width:
add_interface_port mm_slave s0_rdata readdata output 32

width expression:
add_parameter DATA_WIDTH INTEGER 32
add_interface_port s0 rdata readdata output "DATA_WIDTH/2"

Related Information
- add_interface on page 465
- get_port_properties on page 475
• `get_port_property` on page 476
• `get_port_property` on page 476
• `Direction Properties` on page 570
• `Avalon Interface Specifications`
7.1.1.3. get_interfaces

**Description**
Returns a list of top-level interfaces.

**Availability**
Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

**Usage**
get_interfaces

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

**Arguments**
No arguments.

**Example**

```
get_interfaces
```

**Related Information**
add_interface on page 465
7.1.1.4. **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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>The name of a top-level interface.</td>
</tr>
<tr>
<td>assignment</td>
<td>The name of an assignment.</td>
</tr>
</tbody>
</table>

**Example**

```
get_interface_assignment s1 embeddedsw.configuration.isFlash
```

**Related Information**
- [add_interface](#) on page 465
- [get_interface_assignments](#) on page 471
- [get_interfaces](#) on page 469
7.1.1.5. 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 465
- get_interface_assignment on page 470
- get_interfaces on page 469
7.1.1.6. 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 467
- get_port_property on page 476
- set_port_property on page 480
7.1.1.7. 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 474
- set_interface_property on page 479
- Avalon Interface Specifications
7.1.1.8. 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 473
• set_interface_property on page 479
• Avalon Interface Specifications
7.1.1.9. 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 467
- get_port_property on page 476
- set_port_property on page 480
- Port Properties on page 568
7.1.1.10. 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 467
- get_port_properties on page 475
- set_port_property on page 480
- Port Properties on page 568
7.1.1.11. 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 Platform Designer Tools**

There are several assignments that guide behavior in the Platform Designer tools.

- `qsyst.ui.export_name`: If present, this interface should always be exported when an instance is added to a Platform Designer system. The value is the requested name of the exported interface in the parent system.

- `qsyst.ui.connect`: If present, this interface should be auto-connected when an instance is added to a Platform Designer system. The value is a comma-separated list of other interfaces on the same instance that should be connected with this interface.
ui.blockdiagram.direction: If present, the direction of this interface in the block diagram is set by the user. The value is either "output" or "input".

Related Information
- add_interface on page 465
- get_interface_assignment on page 470
- get_interface_assignments on page 471
7. Component Interface Tcl Reference

7.1.1.12. 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 473
- `get_interface_property` on page 474
- *Avalon Interface Specifications*
7.1.1.13. set_port_property

**Description**
Sets a port property.

**Availability**
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 467
- get_port_properties on page 475
- set_port_property on page 480
7.1.1.14. 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 Platform Designer maintains all properties and connections automatically.

**Availability**
Parameter Upgrade

**Usage**

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

**Returns**
No return value.

**Arguments**

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

**Example**

```
set_interface_upgrade_map { avalon_master_interface
new_avalon_master_interface }
```

7.1.2. Parameters

- add_parameter on page 483
- get_parameters on page 484
- get_parameter_properties on page 485
- get_parameter_property on page 486
- get_parameter_value on page 487
- get_string on page 488
- load_strings on page 489
- set_parameter_property on page 490
- set_parameter_value on page 491
- decode_address_map on page 492
7.1.2.1. 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 485
- [get_parameter_property](#) on page 486
- [get_parameter_value](#) on page 487
- [set_parameter_property](#) on page 490
- [set_parameter_value](#) on page 491
- Parameter Type Properties on page 566
7.1.2.2. 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 483
- `get_parameter_property` on page 486
- `get_parameter_value` on page 487
- `get_parameters` on page 484
- `set_parameter_property` on page 490
7.1.2.3. 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 483
- `get_parameter_property` on page 486
- `get_parameter_value` on page 487
- `get_parameters` on page 484
- `set_parameter_property` on page 490
- *Parameter Properties* on page 564
7.1.2.4. 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 483
- get_parameter_properties on page 485
- get_parameter_value on page 487
- get_parameters on page 484
- set_parameter_property on page 490
- set_parameter_value on page 491
- Parameter Properties on page 564
7.1.2.5. get_parameter_value

**Description**
Returns the current value of a parameter defined previously with the `add_parameter` command.

**Availability**
Discovery, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

**Usage**
`get_parameter_value <parameter>`

**Returns**
The value of the parameter.

**Arguments**

*parameter*  The name of the parameter whose value is being retrieved.

**Example**

```
set width [ get_parameter_value fifo_width ]
```

**Notes**
If `AFFECTS_ELABORATION` is false for a given parameter, `get_parameter_value` is not available for that parameter from the elaboration callback. If `AFFECTS_GENERATION` is false then it is not available from the generation callback.

**Related Information**
- `add_parameter` on page 483
- `get_parameter_property` on page 486
- `get_parameters` on page 484
- `set_parameter_property` on page 490
- `set_parameter_value` on page 491
7.1.2.6. 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.

set formatted_string [ format [ get_string TWO_ARGUMENT_MESSAGE_FORMAT ] "arg1" "arg2" ]

Related Information
load_strings on page 489
7.1.2.7. 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*

```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*:

```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:

```properties
TROGDOR = A dragon with a big beefy arm
```

**Related Information**

- `get_string` on page 488
- *Java Properties File*
7.1.2.8. 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 483
- get_parameter_properties on page 485
- set_parameter_property on page 490
- Parameter Properties on page 564
7.1.2.9. 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 is not preserved.

**Availability**
Edit, Elaboration, Validation, Composition, Parameter Upgrade

**Usage**
set_parameter_value <parameter> <value>

**Returns**
No return value.

**Arguments**

*parameter*  The name of the parameter that is being set.

*value*  Specifies the new parameter value.

**Example**

```
set_parameter_value half_clock_rate [ expr { [ get_parameter_value clock_rate ] / 2 } ]
```
7.1.2.10. 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_sram, 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.

```xml
<address-map>
  <slave name='ext_sram' start='0x01000000' end='0x01200000' />
  <slave name='sys_clk_timer' start='0x02120800' end='0x02120820' />
  <slave name='sysid' start='0x021208B8' end='0x021208C0' />
</address-map>
```

**Note:** Intel 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)"
}
```
7.1.3. Display Items

- `add_display_item` on page 494
- `get_display_items` on page 496
- `get_display_item_properties` on page 497
- `get_display_item_property` on page 498
- `set_display_item_property` on page 499
7.1.3.1. add_display_item

Description
Specifies the following aspects of the IP component display:

- Creates logical groups for an IP component's parameters. For example, to create separate groups for the IP component's timing, size, and simulation parameters. An IP component displays the groups and parameters in the order that you specify the display items in the _hw.tcl file.
- Groups a list of parameters to create multi-column tables.
- Specifies an image to provide representation of a parameter or parameter group.
- Creates a button by adding a display item of type action. The display item includes the name of the callback to run.

Availability
Main Program

Usage
add_display_item <parent_group> <id> <type> [ARGS]

Returns

Arguments

parent_group  Specifies the group to which a display item belongs

id  The identifier for the display item. If the item being added is a parameter, this is the parameter name. If the item is a group, this is the group name.

type  The type of the display item. Refer to Display Item Kind Properties.

args (optional)  Provides extra information required for display items.

Example

add_display_item "Timing" read_latency PARAMETER
add_display_item "Sounds" speaker_image_id ICON speaker.jpg
Notes

The following examples illustrate further illustrate the use of arguments:

- `add_display_item groupName id icon path-to-image-file`
- `add_display_item groupName parameterName parameter`
- `add_display_item groupName id text "your-text"`
  
  The `your-text` argument is a block of text that is displayed in the GUI. Some simple HTML formatting is allowed, such as `<b>` and `<i>`, if the text starts with `<html>`.

- `add_display_item parentGroupName childGroupName group [tab]`
  
  The tab is an optional parameter. If present, the group appears in separate tab in the GUI for the instance.

- `add_display_item parentGroupName actionName action buttonClickCallbackProc`

Related Information

- `get_display_item_properties` on page 497
- `get_display_item_property` on page 498
- `get_display_items` on page 496
- `set_display_item_property` on page 499
- `Display Item Kind Properties` on page 572
7.1.3.2. 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 494
- get_display_item_properties on page 497
- get_display_item_property on page 498
- set_display_item_property on page 499
7.1.3.3. 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 494
- get_display_item_property on page 498
- set_display_item_property on page 499
- Display Item Properties on page 571
7.1.3.4. 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 494
- get_display_item_properties on page 497
- get_display_items on page 496
- set_display_item_property on page 499
- Display Item Properties on page 571
7.1.3.5. 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 494
- get_display_item_properties on page 497
- get_display_item_property on page 498
- Display Item Properties on page 571
7.1.4. Module Definition

add_documentation_link on page 501
get_module_assignment on page 502
get_module_assignments on page 503
get_module_ports on page 504
get_module_properties on page 505
get_module_property on page 506
send_message on page 507
set_module_assignment on page 508
set_module_property on page 509
add_hdl_instance on page 510
package on page 511
7.1.4.1. 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

7.1.4.2. 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 503
- set_module_assignment on page 508
7.1.4.3. 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**

```tcl
get_module_assignments
```

**Related Information**
- `get_module_assignment` on page 502
- `set_module_assignment` on page 508
7.1.4.4. 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 465
- add_interface_port on page 467
7.1.4.5. 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 Platform Designer.

**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 506
- set_module_property on page 509
- Module Properties on page 574
7.1.4.6. 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 505
- set_module_property on page 509
- Module Properties on page 574
7.1.4.7. 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 Platform Designer system cannot be generated with existing error messages.
  - **WARNING**--Provides a warning message.
  - **INFO**--Provides an informational message. The INFO level is not available in the Main Program.
  - **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!"
```
7.1.4.8. 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 502
• get_module_assignments on page 503
7.1.4.9. 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 505
- get_module_property on page 506
- Module Properties on page 574
7.1.4.10. add_hdl_instance

Description
Adds an instance of a predefined module, referred to as a child or child instance. The HDL entity generated from this instance can be instantiated and connected within this IP component's HDL.

Availability
Main Program, Elaboration, Composition

Usage
add_hdl_instance <entity_name> <ip_core_type> [<version>]

Returns
The entity name of the added instance.

Arguments

entity_name  Specifies a unique local name that you can use to manipulate the instance. This name is used in the generated HDL to identify the instance.

ip_core_type  The type refers to a kind of instance available in the IP Catalog, for example altera_avalon_uart.

version (optional)  The required version of the specified instance type. If no version is specified, the latest version is used.

Example
add_hdl_instance my_uart altera_avalon_uart

Related Information
- get_instance_parameter_value on page 528
- get_instance_parameters on page 526
- get_instances on page 518
- set_instance_parameter_value on page 531
### 7.1.4.11. package

**Description**

Allows you to specify a particular version of the Platform Designer 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 Platform Designer that you require, such as 14.1.

**Example**

```tcl
package require -exact qsys 14.1
```
7. Component Interface Tcl Reference

7.1.5. Composition

- `add_instance` on page 513
- `add_connection` on page 514
- `get_connections` on page 515
- `get_connection_parameters` on page 516
- `get_connection_parameter_value` on page 517
- `get_instances` on page 518
- `get_instance_interfaces` on page 519
- `get_instance_interface_ports` on page 520
- `get_instance_interface_properties` on page 521
- `get_instance_property` on page 522
- `set_instance_property` on page 523
- `get_instance_properties` on page 524
- `get_instance_interface_property` on page 525
- `get_instance_parameters` on page 526
- `get_instance_parameter_property` on page 527
- `get_instance_parameter_value` on page 528
- `get_instance_port_property` on page 529
- `set_connection_parameter_value` on page 530
- `set_instance_parameter_value` on page 531
7.1.5.1. 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 generates; There is no need to write custom HDL 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 514
• get_instance_interface_property on page 525
• get_instance_parameter_value on page 528
• get_instance_parameters on page 526
• get_instance_property on page 522
• get_instances on page 518
• set_instance_parameter_value on page 531
7.1.5.2. 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 513
- [get_instance_interfaces](#) on page 519
7.1.5.3. 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 514
7.1.5.4. 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 514](#)
- [get_connection_parameter_value on page 517](#)
7.1.5.5. 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 514
• get_connection_parameters on page 516
7.1.5.6. 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 510
- `add_instance` on page 513
- `get_instance_parameter_value` on page 528
- `get_instance_parameters` on page 526
- `set_instance_parameter_value` on page 531
7.1.5.7. 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 513
• get_instance_interface_ports on page 520
• get_instance_interfaces on page 519
7.1.5.8. 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 513
- get_instance_interfaces on page 519
- get_instance_port_property on page 529
7.1.5.9. 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 513
- get_instance_interface_property on page 525
- get_instance_interfaces on page 519
7.1.5.10. 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 513
- get_instance_properties on page 524
- set_instance_property on page 523
- Instance Properties on page 563
7.1.5.11. 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 513
- get_instance_properties on page 524
- get_instance_property on page 522
- Instance Properties on page 563
7.1.5.12. 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 Platform Designer.

**Availability**
Discovery, Main Program, Edit, Elaboration, Validation, Generation, Composition, Fileset Generation, Parameter Upgrade

**Usage**
get_instance_properties

**Returns**
List of strings. Refer to *Instance Properties*.

**Arguments**
No arguments.

**Example**
get_instance_properties

**Related Information**
- add_instance on page 513
- get_instance_property on page 522
- set_instance_property on page 523
- Instance Properties on page 563
7.1.5.13. 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 513
• get_instance_interfaces on page 519
7.1.5.14. 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 510
- add_instance on page 513
- get_instance_parameter_value on page 528
- get_instances on page 518
- set_instance_parameter_value on page 531
7.1.5.15. **get_instance_parameter_property**

**Description**
Returns the value of a property on a parameter in a child instance. Parameter properties are metadata that describe how the Platform Designer tools use the parameter.

**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 513
- Parameter Properties on page 564
7.1.5.16. 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 510
- add_instance on page 513
- get_instance_parameters on page 526
- get_instances on page 518
- set_instance_parameter_value on page 531
7.1.5.17. 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 513
- get_instance_interface_ports on page 520
- Port Properties on page 568
7.1.5.18. 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 514
- `get_connection_parameter_value` on page 517
7.1.5.19. 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 cannot 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 510
- add_instance on page 513
- get_instance_parameter_value on page 528
- get_instances on page 518
7.1.6. Fileset Generation

add_fileset on page 533
add_fileset_file on page 534
set_fileset_property on page 535
get_fileset_file_attribute on page 536
set_fileset_file_attribute on page 537
get_fileset_properties on page 538
get_fileset_property on page 539
get_fileset_sim_properties on page 540
set_fileset_sim_properties on page 541
create_temp_file on page 542
7.1.6.1. add_fileset

**Description**

Adds a generation fileset for a particular target as specified by the `kind`. Platform Designer 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. Platform Designer 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**

```tcl
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**

```tcl
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 534
- get_fileset_property on page 539
- Fileset Properties on page 576
7.1.6.2. add_fileset_file

Description
Adds a file to the generation directory. You can specify source file locations with either an absolute path, or a path relative to the IP component's _hw.tcl file. When you use the add_fileset_file command in a fileset callback, the Intel Quartus Prime software compiles the files in the order that they are added.

Availability
Main Program, Fileset Generation

Usage
add_fileset_file <output_file> <file_type> <file_source> <path_or_contents> [<attributes>]

Returns
No return value.

Arguments

output_file Specifies the location to store the file after Platform Designer generation

file_type The kind of file. Refer to File Kind Properties.

file_source Specifies whether the file is being added by path, or by file contents. Refer to File Source Properties.

path_or_contents When the file_source is PATH, specifies the file to be copied to output_file. When the file_source is TEXT, specifies the text contents to be stored in the file.

attributes (optional) An optional list of file attributes. Typically used to specify that a file is intended for use only in a particular simulator. Refer to File Attribute Properties.

Example

add_fileset_file "./implementation/rx_pma.sv" SYSTEM_VERILOG PATH
synth_rx_pma.sv
add_fileset_file gui.sv SYSTEM_VERILOG TEXT "Customize your IP core"

Related Information
- add_fileset on page 533
- get_fileset_file_attribute on page 536
- File Kind Properties on page 580
- File Source Properties on page 581
- File Attribute Properties on page 579
7.1.6.3. 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 is 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 533
- Fileset Properties on page 576
7.1.6.4. 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 533
• add_fileset_file on page 534
• get_fileset_file_attribute on page 536
• File Attribute Properties on page 579
• add_fileset on page 533
• add_fileset_file on page 534
• get_fileset_file_attribute on page 536
• File Attribute Properties on page 579
7.1.6.5. 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
7.1.6.6. 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 533
- get_fileset_properties on page 538
- set_fileset_property on page 535
- Fileset Properties on page 576
7.1.6.7. 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 576
7.1.6.8. 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 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 533
* set_fileset_sim_properties on page 541
* Operating System Properties on page 588
* Simulator Properties on page 582
7.1.6.9. 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 540
• Operating System Properties on page 588
• Simulator Properties on page 582
7.1.6.10. create_temp_file

**Description**

Creates a temporary file, which you can use inside the fileset callbacks of a _hw.tcl file. This temporary file is included in the generation output if it is added using the add_fileset_file command.

**Availability**

Fileset Generation

**Usage**

create_temp_file <path>

**Returns**

The path to the temporary file.

**Arguments**

*path*  The name of the temporary file.

**Example**

```
set filelocation [create_temp_file "./hdl/compute_frequency.v"]
add_fileset_file compute_frequency.v VERILOG PATH ${filelocation}
```

**Related Information**

- add_fileset on page 533
- add_fileset_file on page 534
7. Component Interface Tcl Reference

7.1.7. Miscellaneous

- `check_device_family_equivalence` on page 544
- `get_device_family_displayname` on page 545
- `get_qip_strings` on page 546
- `set_qip_strings` on page 547
- `set_interconnect_requirement` on page 548
7.1.7.1. 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 545
7.1.7.2. 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 544
7.1.7.3. get_qip_strings

**Description**
Returns a Tcl list of QIP strings for the IP component.

**Availability**
Discovery, Main Program, Edit, Elaboration, Validation, Composition, Parameter Upgrade

**Usage**
get_qip_strings

**Returns**
A Tcl list of qip strings set by this IP component.

**Arguments**
No arguments.

**Example**
```
set strings [ get_qip_strings ]
```

**Related Information**
set_qip_strings on page 547
7.1.7.4. set_qip_strings

Description
Places strings in the Intel Quartus Prime IP File (.qip) file, which Platform Designer passes to the command as a Tcl list. You add the .qip file to your Intel 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 546
7.1.7.5. 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
```
7. Component Interface Tcl Reference

7.1.8. SystemVerilog Interface Commands

- `add_sv_interface` on page 550
- `get_sv_interfaces` on page 551
- `get_sv_interface_property` on page 552
- `get_sv_interface_properties` on page 553
- `set_sv_interface_property` on page 554
7.1.8.1. add_sv_interface

**Description**
Adds a SystemVerilog interface to the IP component.

**Availability**
Elaboration, Global

**Usage**
add_sv_interface `<sv_interface_name>` `<sv_interface_type>`

**Returns**
No return value.

**Arguments**

*sv_interface_name*   The name of the SystemVerilog interface in the IP component.

*sv_interface_type*   The type of the SystemVerilog interface used by the IP component.

**Example**

```
add_sv_interface my_sv_interface my_sv_interface_type
```
7.1.8.2. get\_sv\_interfaces

**Description**
Returns the list of SystemVerilog interfaces in the IP component.

**Availability**
Elaboration, Global

**Usage**
get\_sv\_interfaces

**Returns**

*String[]* Returns the list of SystemVerilog interfaces defined in the IP component.

**Arguments**
No arguments.

**Example**

get\_sv\_interfaces
7.1.8.3. get_sv_interface_property

**Description**
Returns the value of a single SystemVerilog interface property from the specified interface.

**Availability**
Elaboration, Global

**Usage**
get_sv_interface_property <sv_interface_name> <sv_interface_property>

**Returns**

various The property value.

**Arguments**

sv_interface_name The name of a SystemVerilog interface of the system.

sv_interface_property The name of the property. Refer to System Verilog Interface Properties.

**Example**

get_sv_interface_property my_sv_interface USE_ALL_PORTS
7.1.8.4. get_sv_interface_properties

**Description**
Returns the names of all the available SystemVerilog interface properties common to all interface types.

**Availability**
Elaboration, Global

**Usage**
get_sv_interface_properties

**Returns**

`String[]` The list of SystemVerilog interface properties.

**Arguments**
No arguments.

**Example**

```
get_sv_interface_properties
```
7.1.8.5. set_sv_interface_property

**Description**
Sets the value of a property on a SystemVerilog interface.

**Availability**
Elaboration, Global

**Usage**
set_sv_interface_property <sv_interface_name> <sv_interface_property> <value>

**Returns**
No return value.

**Arguments**

- **interface**  The name of a SystemVerilog interface.

- **sv_interface_property**  The name of the property. Refer to *SystemVerilog Interface Properties*.

- **value**  The property value.

**Example**

```
set_sv_interface_property my_sv_interface USE_ALL_PORTS True
```
7.1.9. Wire-Level Expression Commands

set_wirelevel_expression on page 431
get_wirelevel_expressions on page 432
remove_wirelevel_expressions on page 433
7.1.9.1. set_wirelevel_expression

**Description**
Applies a wire-level expression to an optional input port or instance in the system.

**Usage**
set_wirelevel_expression <instance_or_port_bitselection> <expression>

**Returns**
No return value.

**Arguments**

- **instance_or_port_bitselection** Specify the instance or port to which the wire-level expression using the `<instance_name>`.`<port_name>[<bit_selection>]` format. The bit selection can be a bit-select, for example [0], or a partial range defined in descending order, for example [7:0]. If no bit selection is specified, the full range of the port is selected.

- **expression** The expression to be applied to an optional input port.

**Examples**

- `set_wirelevel_expression {module0.portA[7:0]} "8'b0"`
- `set_wirelevel_expression module0.portA "8'b0"
- `set_wirelevel_expression {module0.portA[0]} "1'b0"`
## 7.1.9.2. get_wirelevel_expressions

**Description**
Retrieve a list of wire-level expressions from an optional input port, instance, or all expressions in the current level of system hierarchy. If the port *bit selection* is specified as an argument, the range must be identical to what was used in the `set_wirelevel_expression` statement.

**Usage**
get_wirelevel_expressions `<instance_or_port_bitselection>`

**Returns**
String[] A flattened list of wire-level expressions. Every item in the list consists of right- and left-hand clauses of a wire-level expression. You can loop over the returned list using `foreach(port expr) $return_list{}`.

**Arguments**

*instance_or_port_bitselection* Specifies which instance or port from which a list of wire-level expressions are retrieved using the `<instance_name>.<port_name>[<bit_selection>]` format.

- If no `<port_name>[<bit_selection>]` is specified, the command causes the return of all expressions from the specified instance.
- If no argument is present, the command causes the return of all expressions from the current level of system hierarchy.

The *bit selection* can be a bit-select, for example `[0]`, or a partial range defined in descending order, for example `[7:0]`. If no *bit selection* is specified, the full range of the port is selected.

**Example**

```tcl
get_wirelevel_expressions
get_wirelevel_expressions module0
get_wirelevel_expressions {module0.portA[7:0]}
```
7.1.9.3. remove_wirelevel_expressions

**Description**
Remove a list of wire-level expressions from an optional input port, instance, or all expressions in the current level of system hierarchy. If the port bit selection is specified as an argument, the range must be identical to what was used in the set_wirelevel_expressions statement.

**Usage**
remove_wirelevel_expressions <instance_or_port_bitselection>

**Returns**
No return value.

**Arguments**

*instance_or_port_bitselection* Specifies which instance or port from which a list of wire-level expressions are removed using the `<instance_name>).<port_name>[<bit_selection>]` format.

- If no `<port_name>[<bit_selection>]` is specified, the command causes the removal of all expressions from the specified instance.
- If no argument is present, the command causes the return of all expressions from the current level of system hierarchy.

The *bit selection* can be a bit-select, for example [0], or a partial range defined in descending order, for example [:0]. If no *bit selection* is specified, the full range of the port is selected.

**Examples**

remove_wirelevel_expressions
remove_wirelevel_expressions module0
remove_wirelevel_expressions {module0.portA[7:0]}
7.2. Platform Designer _hw.tcl Property Reference

- Script Language Properties on page 560
- Interface Properties on page 561
- SystemVerilog Interface Properties on page 561
- Instance Properties on page 563
- Parameter Properties on page 564
- Parameter Type Properties on page 566
- Parameter Status Properties on page 567
- Port Properties on page 568
- Direction Properties on page 570
- Display Item Properties on page 571
- Display Item Kind Properties on page 572
- Display Hint Properties on page 573
- Module Properties on page 574
- Fileset Properties on page 576
- Fileset Kind Properties on page 577
- Callback Properties on page 578
- File Attribute Properties on page 579
- File Kind Properties on page 580
- File Source Properties on page 581
- Simulator Properties on page 582
- Port VHDL Type Properties on page 583
- System Info Type Properties on page 584
- Design Environment Type Properties on page 586
- Units Properties on page 587
- Operating System Properties on page 588
- Quartus.ini Type Properties on page 589
### 7.2.1. 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>
### 7.2.2. 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_colorSpaceConverter.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 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 have their base address offset by this amount in the SVD file. |
| SV_INTERFACE            | When SV_INTERFACE is set, all the ports in the given interface are part of the SystemVerilog interface. Example:  
```
set_interface_property my_qsys_interface SV_INTERFACE
```
| IPXACT_REGISTER_MAP     | Specifies the connection point's associated IP-XACT register map file. Platform Designer supports register map files in IP-XACT 2009 or 2014 format. Example:  
```
set_interface_property my_qsys_interface IPXACT_REGISTER_MAP <path_to_ipxact_reg_file>
```
| IPXACT_REGISTER_MAP_VARIABLES | For macro substitution inside the IP-XACT register map file.  
Specifies a list of key value pairs, where key is the macro name and value is the replacement text that substitutes the macros in the IP-XACT register map. |

**Related Information**

Interfaces and Ports on page 464

### 7.2.3. SystemVerilog Interface Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV_INTERFACE_TYPE</td>
<td>Set the interface type of the SystemVerilog interface.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USE_ALL_PORTS</td>
<td>When USE_ALL_PORTS is set to true, all the ports defined in the Module, are declared in this SystemVerilog interface. USE_ALL_PORTS must be set to true only if the module has one SystemVerilog interface and the SystemVerilog interface signal names match with the port names declared for Platform Designer interface. When USE_ALL_PORTS is true, SV_INTERFACE_PORT or SV_INTERFACE_SIGNAL port properties should not be set.</td>
</tr>
</tbody>
</table>
### 7.2.4. Instance Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLINSTANCE_GET_GENERATED_NAME</td>
<td>Platform Designer 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>
</tbody>
</table>
| String[]   | DISPLAY_HINT        | Provides a hint about how to display a property. The following values are possible:  
|             |                     |  • 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—for integer parameters, display and interpret the value as a hexadecimal number, for example: 0x00000010 instead of 16.  
<p>|             |                     |  • fixed_size—for string_list and integer_list parameters, the fixed_size DISPLAY_HINT eliminates the add and remove buttons from tables. |
| String     | DISPLAY_NAME        | This is the GUI label that appears to the left of this parameter.                                                                                                                                             |
| String     | DISPLAY_UNITS       | This is the GUI label that appears to the right of the parameter.                                                                                                                                             |</p>
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 not explicitly set a parameter value, and use the DEFAULT_VALUE property. The practical result is that older instances continue to use DEFAULT_VALUE for the parameter and new instances use the value that NEW_INSTANCE_VALUE assigns.</td>
</tr>
<tr>
<td>String</td>
<td>SV_INTERFACE_PARAMETER</td>
<td>This parameter is used in the SystemVerilog interface instantiation. Example: set_parameter_property my_parameter SV_INTERFACE_PARAMETER my_sv_interface</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 584
- Parameter Type Properties on page 566
- Units Properties on page 587
### 7.2.6. 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 true or false.</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 WIDTH 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>
### 7.2.7. 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>
### 7.2.8. 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 is 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 Platform Designer 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 Platform Designer 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 Platform Designer 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>String</td>
<td>SV_INTERFACE_PORT</td>
<td>This port from the module is used as I/O in the SystemVerilog interface instantiation. The top-level wrapper of the module which contains this port is from the SystemVerilog interface. Example: set_port_property port_x SV_INTERFACE_PORT my_sv_interface</td>
</tr>
<tr>
<td>String</td>
<td>SV_INTERFACE_PORT_NAME</td>
<td>This property is used only when the Platform Designer port name defined for the module is different from the port name in the SystemVerilog interface. Example: set_port_property port_x SV_INTERFACE_PORT_NAME port_a When writing the RTL, the Platform Designer port name port_x is mapped to RTL name port_a in the SystemVerilog interface</td>
</tr>
<tr>
<td>String</td>
<td>SV_INTERFACE_SIGNAL</td>
<td>This port from the module is assumed to be inside the SystemVerilog interface or the modport used by the module. The top-level wrapper of the module containing this port is unwrapped from SystemVerilog interface. Example: set_port_property port_y SV_INTERFACE_SIGNAL my_sv_interface</td>
</tr>
<tr>
<td>String</td>
<td>SV_INTERFACE_SIGNAL_NAME</td>
<td>This property is only used when the Platform Designer port name defined for the module is different from the port name in the SystemVerilog interface. Example: set_port_property port_y SV_INTERFACE_SIGNAL_NAME port_b</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>TERMINATION</td>
<td>When true, instead of connecting the port to the Platform Designer 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 570
- Port VHDL Type Properties on page 583
- Avalon Interface Specifications
### 7.2.9. 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>
### 7.2.10. 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>
### 7.2.11. 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>
### 7.2.12. 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>
### 7.2.13. 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 Intel 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 Intel 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 not 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 implements a half-rate bridge.&quot;</td>
</tr>
<tr>
<td>DISPLAY_NAME</td>
<td>The name to display when referencing the IP component, such as &quot;My Platform Designer 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 creates 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 software tools. When true, the children's address are not visible. When false, the children's addresses are visible.</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>
</tbody>
</table>
### Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCTURAL_COMPOSITION_CALLBACK</td>
<td>The name of the structural composition callback. This callback is used to represent the structural hierarchal 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>
## 7.2.14. 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 TOP_LEVEL name, and the HDL must not be parameterized. Platform Designer runs the generate callback one time, regardless of the number of instances in the system.</td>
</tr>
</tbody>
</table>
### 7.2.15. Fileset Kind Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE_DESIGN</td>
<td>Contains example design files.</td>
</tr>
<tr>
<td>QUARTUS_SYNTH</td>
<td>Contains files that Platform Designer uses for the Intel Quartus Prime software synthesis.</td>
</tr>
<tr>
<td>SIM_VERILOG</td>
<td>Contains files that Platform Designer uses for Verilog HDL simulation.</td>
</tr>
<tr>
<td>SIM_VHDL</td>
<td>Contains files that Platform Designer uses for VHDL simulation.</td>
</tr>
<tr>
<td>SYSTEMVERILOG_INTERFACE</td>
<td>This file is treated as SystemVerilog interface file by the Platform Designer. Example:</td>
</tr>
<tr>
<td></td>
<td>add_fileset_file mem_ifc.sv SYSTEM_VERILOG PATH &quot;.ifc/mem_ifc.sv&quot; SYSTEMVERILOG_INTERFACE</td>
</tr>
</tbody>
</table>
### 7.2.16. 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>
### 7.2.17. 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>
### 7.2.18. 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>SystemVerilog HDL</td>
</tr>
<tr>
<td>SYSTEM_VERILOG_ENCRYPT</td>
<td>Encrypted SystemVerilog HDL</td>
</tr>
<tr>
<td>SYSTEM_VERILOG_INCLUDE</td>
<td>SystemVerilog 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>
## 7.2.19. 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 <code>output_file</code>.</td>
</tr>
<tr>
<td>TEXT</td>
<td>Specifies an arbitrary text string for the contents of <code>output_file</code>.</td>
</tr>
</tbody>
</table>
### 7.2.20. 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>
</tbody>
</table>
7.2.21. Port VHDL Type Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 are 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>
### 7.2.22. 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. Platform Designer 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 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 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>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>
</tbody>
</table>
### Type 

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 contains information about the slave, the 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 586
7.2.23. 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 Platform Designer interfaces.</td>
</tr>
</tbody>
</table>
### 7.2.24. 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>
## 7.2.25. 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>WINDOWS32</td>
<td>Windows 32-bit</td>
</tr>
<tr>
<td>WINDOWS64</td>
<td>Windows 64-bit</td>
</tr>
</tbody>
</table>
### 7.2.26. 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>
7.3. Component Interface Tcl Reference Revision History

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

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2017.11.06        | 17.1.0                      | • Changed instances of Qsys to Platform Designer (Standard)  
                        • Added statement clarifying use of brackets. |
| 2015.11.02        | 15.1.0                      | Changed instances of Quartus II to Quartus Prime. |
| 2015.05.04        | 15.0.0                      | Edit to add_fileset_file command. |
| December 2014     | 14.1.0                      | • set_interface_upgrade_map  
                        • Moved Port Roles (Interface Signal Types) section to Qsys Interconnect. |
| November 2013     | 13.1.0                      | • add_hdl_instance |
| May 2013          | 13.0.0                      | • Consolidated content from other Qsys chapters.  
                        • Added AMBA APB support. |
| November 2012     | 12.1.0                      | • Added the demo_axi_memory example with screen shots and example._hw.tcl code. |
| June 2012         | 12.0.0                      | • Added AXI 3 support.  
                        • Added: set_display_item_property,  
                        set_parameter_property, LONG_DESCRIPTION, and static filesets. |
| November 2011     | 11.1.0                      | • Template update.  
                        • Added: set_qip_strings, get_qip_strings,  
                        get_device_family_displayname,  
                        check_device_family_equivalence. |
| May 2011          | 11.0.0                      | • Revised section describing HDL and composed component implementations.  
                        • Separated reset and clock interfaces in example.  
                        • Added: TRISTATECONDUIT_INFO, GENERATION_ID, UNIQUE_ID  
                        SYSTEM_INFO.  
                        • Added: WIDTH and SYSTEM_INFO_ARG parameter properties.  
                        • Removed the doc_type argument from the  
                        add_documentation_link command.  
                        • Removed: get_instance_parameter_properties  
                        • Added: add_fileset, add_fileset_file, create_temp_file.  
                        • Updated Tcl examples to show separate clock and reset interfaces. |
| December 2010     | 10.1.0                      | • Initial release. |

Related Information

Documentation Archive

For previous versions of the Intel Quartus Prime Handbook, search the documentation archives.
A. Intel Quartus Prime Standard Edition User Guides

Refer to the following user guides for comprehensive information on all phases of the Intel Quartus Prime Standard Edition FPGA design flow.

Related Information

  Introduces the basic features, files, and design flow of the Intel Quartus Prime Standard Edition software, including managing Intel Quartus Prime Standard Edition projects and IP, initial design planning considerations, and project migration from previous software versions.

  Describes creating and optimizing systems using Platform Designer, a system integration tool that simplifies integrating customized IP cores in your project. Platform Designer automatically generates interconnect logic to connect intellectual property (IP) functions and subsystems.

  Describes best design practices for designing FPGAs with the Intel Quartus Prime Standard Edition software. HDL coding styles and synchronous design practices can significantly impact design performance. Following recommended HDL coding styles ensures that Intel Quartus Prime Standard Edition synthesis optimally implements your design in hardware.

  Describes set up, running, and optimization for all stages of the Intel Quartus Prime Standard Edition Compiler. The Compiler synthesizes, places, and routes your design before generating a device programming file.

  Describes Intel Quartus Prime Standard Edition settings, tools, and techniques that you can use to achieve the highest design performance in Intel FPGAs. Techniques include optimizing the design netlist, addressing critical chains that limit retiming and timing closure, and optimization of device resource usage.

  Describes operation of the Intel Quartus Prime Standard Edition Programmer, which allows you to configure Intel FPGA devices, and program CPLD and configuration devices, via connection with an Intel FPGA download cable.

  Describes Partial Reconfiguration, an advanced design flow that allows you to reconfigure a portion of the FPGA dynamically, while the remaining FPGA design continues to function. Define multiple personas for a particular design region, without impacting operation in other areas.
  Describes RTL- and gate-level design simulation support for third-party simulation tools by Aldec*, Cadence*, Mentor Graphics*, and Synopsys that allow you to verify design behavior before device programming. Includes simulator support, simulation flows, and simulating Intel FPGA IP.

  Describes support for optional synthesis of your design in third-party synthesis tools by Mentor Graphics*, and Synopsys. Includes design flow steps, generated file descriptions, and synthesis guidelines.

  Describes a portfolio of Intel Quartus Prime Standard Edition in-system design debugging tools for real-time verification of your design. These tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. These tools include System Console, Signal Tap logic analyzer, Transceiver Toolkit, In-System Memory Content Editor, and In-System Sources and Probes Editor.

  Explains basic static timing analysis principals and use of the Intel Quartus Prime Standard Edition Timing Analyzer, 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.

  Describes the Intel Quartus Prime Standard Edition Power Analysis tools that allow accurate estimation of device power consumption. Estimate the power consumption of a device to develop power budgets and design power supplies, voltage regulators, heat sink, and cooling systems.

  Describes timing and logic constraints that influence how the Compiler implements your design, such as pin assignments, device options, logic options, and timing constraints. Use the Pin Planner to visualize, modify, and validate all I/O assignments in a graphical representation of the target device.

  Describes support for optional third-party PCB design tools by Mentor Graphics* and Cadence*. Also includes information about signal integrity analysis and simulations with HSPICE and IBIS Models.

  Describes use of Tcl and command line scripts to control the Intel Quartus Prime Standard Edition software and to perform a wide range of functions, such as managing projects, specifying constraints, running compilation or timing analysis, or generating reports.