Creating Qsys Components 7

2013.11.4

QII51022

Subscribe Send Feedback

In order to describe and package IP components for use in a Qsys system, you must create a Hardware Component Definition File (**_hw.tcl**) which will describes your component, its interfaces and HDL files. Qsys provides the Component Editor to help you create a simple **_hw.tcl** file.

The **Demo AXI Memory** example on the **Qsys Design Examples** page of the Altera® web site provides the full code examples that appear in the following topics.

Qsys supports standard Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), and AMBA APB™ 3 (version 1.0) interface specifications.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specifications
- Demo AXI Memory Example

Qsys Components

A Qsys component includes the following elements:

- Information about the component type, such as name, version, and author.
- HDL description of the component's hardware, including SystemVerilog, Verilog HDL, or VHDL files
- Constraint files (Synopsys Design Constraints File (**.sdc**) and/or Quartus II IP File (**.qip**)) that define the component for synthesis and simulation.
- A component's interfaces, including I/O signals.
- The parameters that configure the operation of the component.

Component Interface Support

Components can have any number of interfaces in any combination. Each interface represents a set of signals that you can connect within a Qsys system, or export outside of a Qsys system.

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

ISO 9001:2008 Registered

101 Innovation Drive, San Jose, CA 95134

Qsys components can include the following types of interfaces:

- **Memory-Mapped**—Implements a partial crossbar interconnect structure (Avalon-MM, AXI, and APB) that provides concurrent paths between master and slaves. Interconnect consists of synchronous logic and routing resources inside the FPGA, and implementation is based on a network-on-chip architecture.
- **Streaming**—Connects Avalon Streaming (Avalon-ST) sources and sinks that stream unidirectional data, as well as high-bandwidth, low-latency components. Streaming creates datapaths for unidirectional traffic, including multichannel streams, packets, and DSP data. The Avalon-ST interconnect is flexible and can implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet, Interlaken, and video. You can define bus widths, packets, and error conditions.
- **Interrupts**—Connects interrupt senders and the interrupt receivers of the component that serves them. Qsys supports individual, single-bit interrupt requests (IRQs). In the event that multiple senders assert their IRQs simultaneously, the receiver logic (typically under software control) determines which IRQ has highest priority, then responds appropriately.
- **Clocks**—Connects clock output interfaces with clock input interfaces. Clock outputs can fan-out without the use of a bridge. A bridge is required only when a clock from an external (exported) source connects internally to more than one source.
- **Resets**—Connects reset sources with reset input interfaces. If your system requires a particular positive-edge or negative-edge synchronized reset, Qsys inserts a reset controller to create the appropriate reset signal. If you design a system with multiple reset inputs, the reset controller ORs all reset inputs and generates a single reset output.
- **Conduits**—Connects point-to-point conduit interfaces, or represent signals that are exported from the Qsys system. Qsys uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Qsys system as a point-to-point connection, or conduit interfaces can be exported and brought to the top-level of the system as top-level system I/O. You can use conduits to connect to external devices, for example external DDR SDRAM memory, and to FPGA logic defined outside of the Qsys system.

Component Structure

Altera provides components automatically installed with the Quartus® II software. You can obtain a list of Qsys-compliant components provided by third-party IP developers on Altera's **Intellectual Property & Reference Designs** page by typing: **qsys certified** in the **Search** box, and then selecting **IP Core & Reference Designs**. Components are also provided with Altera development kits, which are listed on the **All Development Kits** page.

Every component is defined with a < *component_name* >**_hw.tcl file**, a text file written in the Tcl scripting language that describes the component to Qsys. When you design your own custom component, you can create the **_hw.tcl file** manually, or by using the Qsys Component Editor.

The Component Editor simplifies the process of creating **_hw.tcl** files by creating a file that you can edit outside of the Component Editor to add advanced procedures. When you edit a previously saved **_hw.tcl** file, Qsys automatically backs up the earlier version as **_hw.tcl~**.

You can move component files into a new directory, such as a network location, so that other users can use the component in their systems. The **_hw.tcl** file contains relative paths to the other files, so if you move an **_hw.tcl** file, you should also move all the HDL and other files associated with it.

There are three component types:

- **Static**— Static components always generate the same output, regardless of their parameterization. Components that instantiate static components must have only static children.
- **Generated**—A generated component's fileset callback allows an instance of the component to create unique HDL design files based on the instance's parameter values.
- **Composed**—Composed components are subsystems constructed from instances of other components. You can use a composition callback to manage the subsystem in a composed component.

Related Information

- **Intellectual Property & Reference Designs**
- **Creating a Composed Component or Subsystem** on page 7-28
- **Adding Component Instances to a Static or Generated Component** on page 7-32

Component File Organization

A typical component uses the following directory structure where the names of the directories are not significant:

<component_directory>/

- <***hdl***>/—Contains the component HDL design files, for example **.v**, **.sv**, or **.vhd** files that contain the top-level module, along with any required constraint files.
- <***component_name***> **_hw.tcl**—The component description file.
- <***component_name***> **_sw.tcl**—The software driver configuration file. This file specifies the paths for the **.c** and **.h** files associated with the component, when required.
- <***software***>/—Contains software drivers or libraries related to the component.

Note: Refer to the *Nios II Software Developer's Handbook* for information about writing a device driver or software package suitable for use with the Nios II processor.

Related Information

- **Hardware Abstraction LayerTool Reference (Nios II Software Developer's Handbook)**
- **Nios II Software Build Tool Reference (Nios II Software Developer's Handbook)**

Component Versions

Qsys systems support multiple versions of the same component within the same system; you can create and maintain multiple versions of the same component.

If you have multiple **_hw.tcl** files for components with the same NAME module properties and different VERSION module properties, both versions of the component are available.

If multiple versions of the component are available in the Qsys Library, you can add a specific version of a component by right-clicking the component, and then selecting **Add version** *<version_number>*.

Upgrading IP Components to the Latest Version

When you open a Qsys design, if Qsys detects IP components that require regeneration, the **Upgrade IP Cores** dialog box appears and allows you to upgrade outdated components.

Components that you must upgrade in order to successfully compile your design appear in red. Status icons indicate whether a component is currently being regenerated, the component is encrypted, or that there is not enough information to determine the status of component. To upgrade a component, in the **Upgrade IP Cores** dialog box, select the component that you want to upgrade, and then click **Upgrade**. The Quartus II software maintains a list of all IP components associated with your design on the **Components** tab in the Project Navigator.

Related Information

Upgrade IP Components Dialog Box

Life Cycle of a Component

When you define a component with the Qsys Component Editor, or a custom **_hw.tcl** file, you specify the information that Qsys requires to instantiate the component in a Qsys system and to generate the appropriate output files for synthesis and simulation.

The following phases describe the process when working with components in Qsys:

- **Discovery**—During the discovery phase, Qsys reads the **_hw.tcl** file to identify information that appears in the Qsys Library, such as the component's name, version, and documentation URLs. Each time you open Qsys, the tool searches for the following file types using the default search locations and entries in the **IP Search Path**:
  - **_hw.tcl** files—Each **_hw.tcl** file defines a single component.
  - IP Index (**.ipx**) files—Each **.ipx** file indexes a collection of available components, or a reference to other directories to search.
- **Static Component Definition**—During the static component definition phase, Qsys reads the **_hw.tcl** file to identify static parameter declarations, interface properties, interface signals, and HDL files that define the component. At this stage of the life cycle, the component interfaces might be only partially defined.
- **Parameterization**—During the parameterization phase, after an instance of the component is added to a Qsys system, the user of the component specifies parameters with the component’s parameter editor.
- **Validation**—During the validation phase, Qsys validates the values of each instance's parameters against the allowed ranges specified for each parameter. You can use callback procedures that run during the validation phase to provide validation messages. For example, if there are dependencies between parameters where only certain combinations of values are supported, you can report errors for the unsupported values.

- **Elaboration**—During the elaboration phase, Qsys queries the component for its interface information. Elaboration is triggered when an instance of a component is added to a system, when its parameters are changed, or when a system property changes. You can use callback procedures that run during the elaboration phase to dynamically control interfaces, signals, and HDL files based on the values of parameters. For example, interfaces defined with static declarations can be enabled or disabled during elaboration. When elaboration is complete, the component's interfaces and design logic must be completely defined.
- **Composition**—During the composition phase, a component can manipulate the instances in the component's subsystem. The **_hw.tcl** file uses a callback procedure to provide parameterization and connectivity of subcomponents.
- **Generation**—During the generation phase, Qsys generates synthesis or simulation files for each component in the system into the appropriate output directories, as well as any additional files that support associated tools.

Creating Qsys Components in the Component Editor

The Qsys Component Editor, accessed by clicking **New Component** in the Qsys Library, allows you to create and package a component for use in Qsys. When you use the Component Editor to define a component, the Component Editor writes the information to the **_hw.tcl** file.

The Component Editor allows you to perform the following tasks:

- Specify component’s identifying information, such as name, version, author, etc.
- Specify the SystemVerilog, Verilog HDL, or VHDL files, and constraint files that define the component for synthesis and simulation.
- Create an HDL template for a component by first defining its parameters, signals, and interfaces.
- Associate and define signals for a component’s interfaces.
- Set parameters on interfaces, which specify characteristics.
- Specify relationships between interfaces.
- Declare parameters that alter the component structure or functionality.

If the component is HDL-based, you must define the parameters and signals in the HDL file, and cannot add or remove them in the Component Editor. If you have not yet created the top-level HDL file, you declare the parameters and signals in the Component Editor, and they are then included in the HDL template file that Qsys creates.

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

If you are creating the component using an existing HDL file, the order in which the tabs appear in the Component Editor reflects the recommended design flow for component development. You can use the **Prev** and **Next** buttons at the bottom of the Component Editor window to guide you through the tabs.

If the component is not based on an existing HDL file, enter the parameters, signals, and interfaces first, and then return to the **Files** tab to create the top-level HDL file template. When you click **Finish**, Qsys creates the component **_hw.tcl** file with the details provided on the Component Editor tabs.

After the component is saved, it is available in the Qsys Library.

If you require features in the component that are not supported by the Component Editor, such as callback procedures, you can use the Component Editor to create the **_hw.tcl** file, and then manually edit the file to

complete the component definition. Subsequent topics document the **_hw.tcl** commands that are generated by the Component Editor, as well as some of the advanced features that you can add with your own **_hw.tcl** commands.

Related Information

Component Interface Tcl Reference

Saving a Component and Creating an _hw.tcl File

You save a component by clicking **Finish** in the Component Editor. The Component Editor saves the component to a file with the file name <*component_name*> **_hw.tcl**.

Altera recommends that you save **_hw.tcl** files and their associated files in an **ip/** <*class-name*> directory within your Quartus II project directory. You can also publish component information for use by software, such as a C compiler and a board support package (BSP) generator.

Refer to *Creating a System with Qsys* for information on how to search for and add components to the Qsys library for use in your designs.

Related Information

Publishing Component Information to Embedded Software (Nios II Software Developer's Handbook)

Creating a System with Qsys

Editing a Component with the Component Editor

In Qsys, you make changes to a component by right-clicking the component in the Library, and then clicking **Edit**. After making changes, click **Finish** to save the changes to the **_hw.tcl** file. You can open the **_hw.tcl** file in a text editor to view the hardware Tcl for the component. If you edit the **_hw.tcl** file to customize the component with advanced features, you cannot use the Component Editor to make further changes without over-writing your customized file.

You cannot use the Component Editor to edit components installed with the Quartus II software, such as Altera-provided components. If you edit the HDL for a component and change the interface to the top-level module, you must edit the component to reflect the changes you made to the HDL.

Related Information

Creating Qsys Components (Quartus II Help)

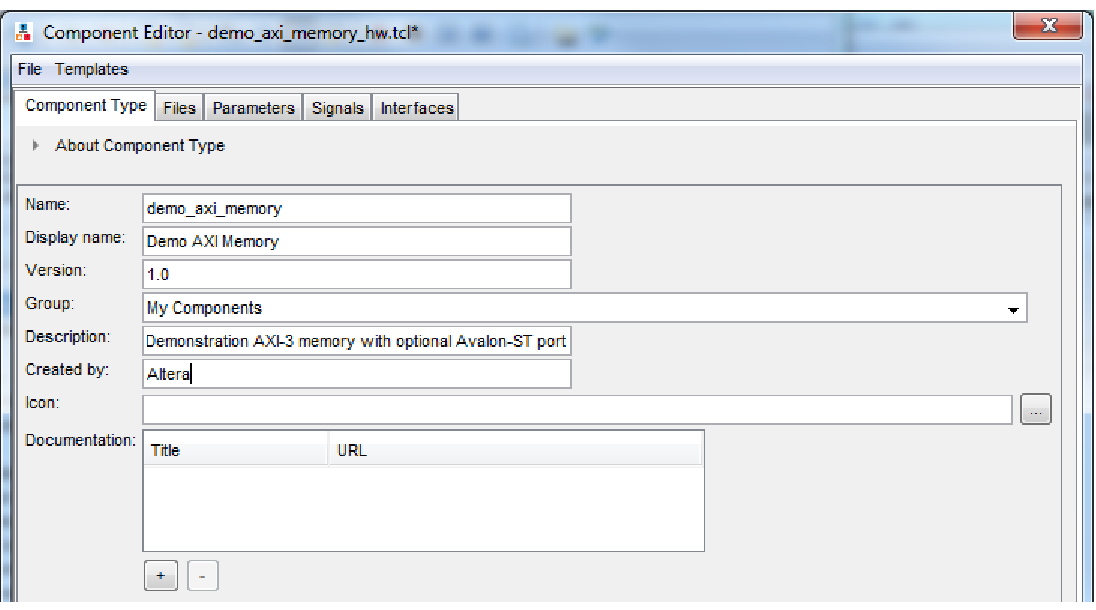
Specifying Basic Component Information

The **Component Type** tab in the 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 Library 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 Library. 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 Library 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 Library in the group you specified under **Project**. Alternatively, if you save the component in the Quartus II installation directory, the component appears in the specified group under **Library**.
- **Description**—Allows you to describe the component. This description appears when the user views the component details.
- **Created By**—Allows you to specify the author of the component.
- **Icon**—Allows you to enter the relative path to an icon file (**.gif**, **.jpg**, or **.png** format) that represents the component and appears as the header in the parameter editor for the component. The default image is the Altera MegaCore function icon.
- **Documentation**—Allows you to add links to documentation for the component, and appears when you right-click the component in the Library, and then select **Details**.
  - To specify an Internet file, begin your path with **http://**, for example: **http://mydomain.com/datasheets/my_memory_controller.html**.
  - To specify a file in the file system, begin your path with **file:///** for Linux, and **file:////** for Windows; for example (Windows): **file:////company_server/datasheets my_memory_controller.pdf**.

The **Display name**, **Group**, **Description**, **Created By**, **Icon**, and **Documentation** entries are optional. Figure 7-1 shows the **Component Type** tab with the component information.

Figure 7-1: Component Type Tab in the Component Editor



When you use the Component Editor to create a component, it writes this basic component information in the **_hw.tcl file**. The example below shows the component hardware Tcl code related to the entries for the **Component Type** tab in figure above. The `package require` command specifies the Quartus II software version that Qsys uses to create the **_hw.tcl** file, and ensures compatibility with this version of the Qsys API in future ACDS releases.

The component defines its basic information with various module properties using the `set_module_property` command. For example, `set_module_property NAME` specifies the name of the component, while `set_module_property VERSION` allows you to specify the version of the component. When you apply a version to the **_hw.tcl** file, it allows the file to behave exactly the same way in future releases of the Quartus II software.

Example 7-1: _hw.tcl Created from Entries in the Component Type tab

```
# request TCL package from ACDS 13.1

package require -exact qsys 13.1

# 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

Specifying Files for Synthesis and Simulation

The **Files** tab in the Component Editor allows you to specify files for synthesis and simulation. If you already have HDL code that describes the Qsys component that you want to create, you can specify the files on the **Files** tab. If you have not yet created the HDL code that describes the component, but you have identified the signals and parameters that you want in the component, you can use the **Files** tab to create a top-level HDL template file. The Component Editor generates the appropriate **_hw.tcl** commands to specify the files. You can also write your own **hw.tcl** file with the same commands, if you are not using the Component Editor.

A component uses filesets to specify the different sets of files that can be generated for an instance of the component. The supported fileset types are: `QUARTUS_SYNTH`, for synthesis and compilation in the Quartus II software, `SIM_VERILOG`, for Verilog HDL simulation, and `SIM_VHDL`, for VHDL simulation.

In a **_hw.tcl** file, you add a fileset to a component with the `add_fileset` command. You then list specific files with the `add_fileset_file` command, which adds the specified files to the most recently declared fileset. The `add_fileset_property` command allows you to add properties such as `TOP_LEVEL`, which specifies the top-level HDL module for the component.

You can populate a fileset with a a fixed list of files, add different files based on a parameter value, or even generate an HDL file with a custom HDL generator function outside of the **_hw.tcl** file.

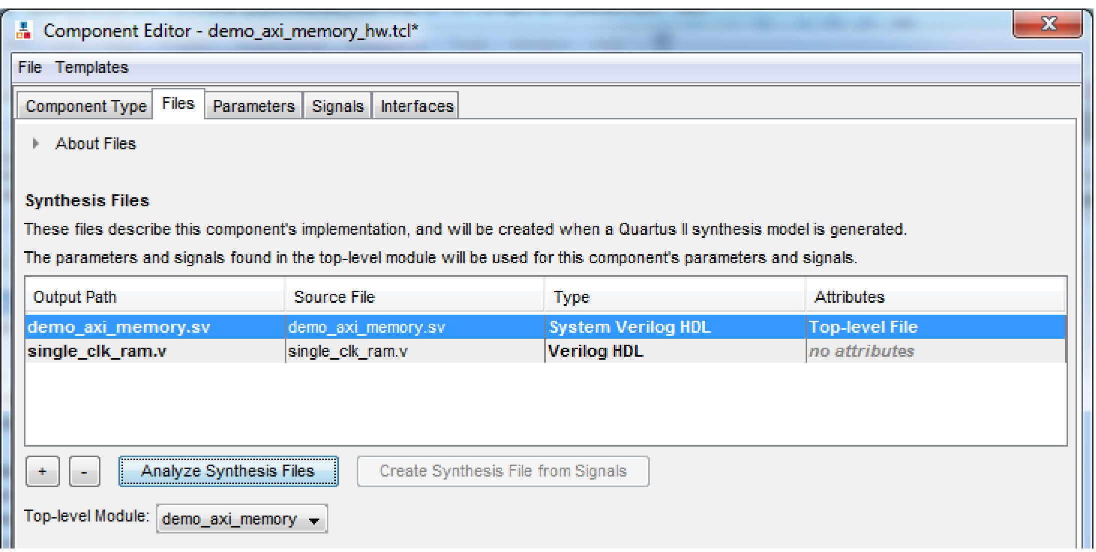
Specifying HDL Files for Synthesis

In the Component Editor, you can add HDL files and other support files that should be included when this component is created to the list of **Synthesis Files** by clicking **+**, and then selecting the files in the **Open** dialog box.

A component must specify an HDL file as the top-level file, which contains the top-level module. The **Synthesis Files** list might also include supporting HDL files, such as timing constraints, or other files required to successfully synthesize and compile in the Quartus II software. The synthesis files for a component are copied to the generation output directory during Qsys system generation.

Figure 7-2 indicates the **demo_axi_memory.sv** file as the top-level file for the component in the **Synthesis Files** section on the **Files** tab.

Figure 7-2: Using HDL Files to Define a Component



Creating a New HDL File for Synthesis

If you do not already have an HDL implementation of the component, you can use the Component Editor to define the component, and then create a simple top-level synthesis file containing the signals and parameters for the component. You can then edit this HDL file to add the logic that directs the component's behavior.

To begin, you first specify the information about the component on the **Parameters**, **Signals**, and **Interfaces** tabs. Then, you return to the **Files** tab to create an HDL file by clicking **Create Synthesis File from Signals**. The Component Editor creates an HDL file from the specified parameters and signals.

Analyzing Synthesis Files

After the top-level HDL file is specified in the Component Editor, click **Analyze Synthesis Files** to analyze the parameters and signals in the top-level, and then select the top-level module from the **Top Level Module** list. If there is a single module or entity in the HDL file, Qsys automatically populates the **Top-level Module** list.

Once analysis is complete and the top-level module is selected, the parameters and signals found in the top-level module are used as the parameters and signals for the component, and you can view them on the **Parameters** and **Signals** tabs. The Component Editor might 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 Component Editor is launched and the **_hw.tcl** is located, you can use standard fixed or relative path notation to identify the file location for the `PATH` variable.

Example 7-2 shows the component hardware Tcl code related to the entries for the **Files Type** tab in the **Synthesis Files** section shown in Figure 7-2.

Example 7-2: _hw.tcl Created from Entries in the Files tab in the Synthesis Files Section

```
# file sets

add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL demo_axi_memory

add_fileset_file demo_axi_memory.sv
SYSTEM_VERILOG PATH demo_axi_memory.sv

add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v
```

Related Information

- Component Interface Tcl Reference
- Specifying HDL Files for Synthesis on page 7-9

Naming HDL Signals for Automatic Interface and Type Recognition

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 this auto-recognition feature, 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. 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 **Interfaces** tab in the Component Editor.

Table 7-1: Interface Type Prefixes for Automatic Signal Recognition

Interface Prefix	Interface Type
asi	Avalon-ST sink (input)
aso	Avalon-ST source (output)
avm	Avalon-MM master
avs	Avalon-MM slave
axm	AXI master

Interface Prefix	Interface Type
axs	AXI slave
apm	APB master
aps	APB slave
coe	Conduit
csi	Clock Sink (input)
cso	Clock Source (output)
inr	Interrupt receiver
ins	Interrupt sender
ncm	Nios II custom instruction master
ncs	Nios II custom instruction slave
rsi	Reset sink (input)
rso	Reset source (output)
tcm	Avalon-TC master
tcs	Avalon-TC slave

Refer to the *Avalon Interface Specifications* or the *AMBA Protocol Specification* for the signal types available for each interface type.

Related Information

- **Avalon Interface SpecificationsProtocol Specification**
- **AMBA Protocol Specification**

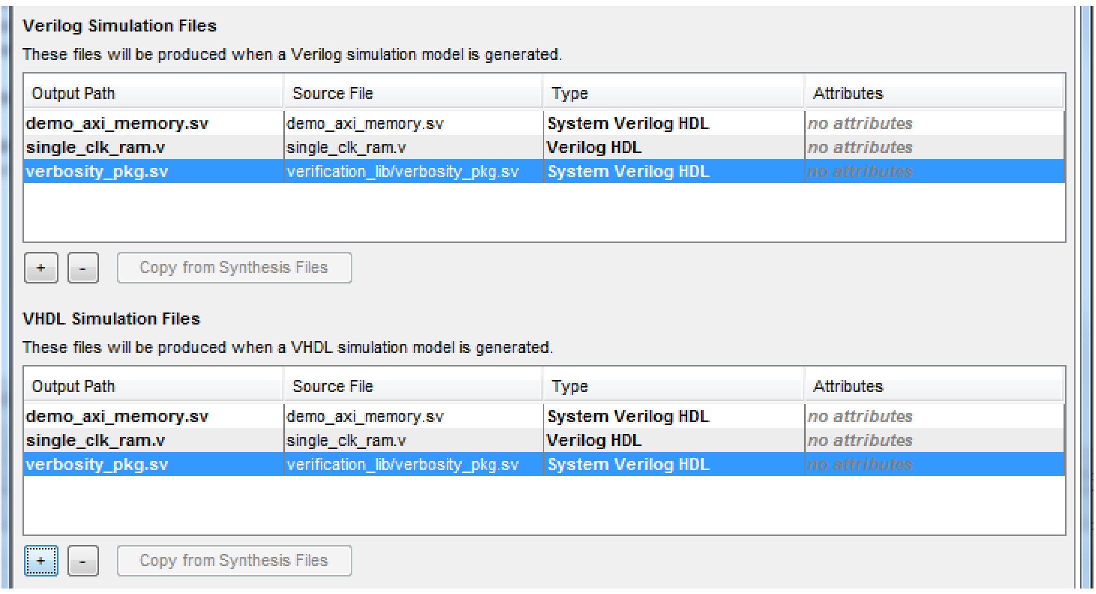
Specifying Files for Simulation

To support Qsys system generation for simulation, a component must specify the VHDL or Verilog simulation files. Simulation files are generated when a user adds the component to a Qsys system and chooses 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 in the Component Editor, on the **Files** tab, click **Copy From Synthesis Files** to copy the list of synthesis files to the **Verilog Simulation Files** or **VHDL Simulation Files** lists.

You specify the simulation files in a similar way as the synthesis files with the fileset commands in a **_hw.tcl** file. The code example below shows `SIM_VERILOG` and `SIM_VHDL` filesets for Verilog and VHDL simulation output files. In this example, the same Verilog files are used for both Verilog and VHDL outputs, and there is one additional System Verilog file added. This method works for designers of Verilog IP to

support users who want to generate a VHDL top-level simulation file when they have a mixed-language simulation tool and license that can read the Verilog output for the component. Figure 7-3 shows the files specified for simulation on the **Files** tab.

Figure 7-3: Specifying the Simulation Output Files



Example 7-3: _hw.tcl Created from Entries in the Files tab in the Simulation Files Section

```
add_fileset SIM_VERILOG SIM_VERILOG "" ""
set_fileset_property SIM_VERILOG TOP_LEVEL demo_axi_memory
add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv

add_fileset_file demo_axi_memory.sv SYSTEM_VERILOG PATH \
demo_axi_memory.sv

add_fileset SIM_VHDL SIM_VHDL "" ""
set_fileset_property SIM_VHDL TOP_LEVEL demo_axi_memory
set_fileset_property SIM_VHDL ENABLE_RELATIVE_INCLUDE_PATHS false

add_fileset_file demo_axi_memory.sv SYSTEM_VERILOG PATH \
demo_axi_memory.sv

add_fileset_file single_clk_ram.v VERILOG PATH single_clk_ram.v

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv
```

Related Information

Component Interface Tcl Reference

Including Internal Register Map Description in the .svd for Slave Interfaces Connected to an HPS Component

Qsys supports the ability for IP component designers to specify register map information on their slave interfaces. This allows components with slave interfaces that are connected to an HPS component to include their internal register description in the generated **.svd** file.

To specify their internal register map, the IP component designer must write and generate their own **.svd** file and attach it to the slave interface using the following command:

`set_interface_property` <*slave interface*> `CMSIS_SVD_FILE` <*file path*>

The `CMSIS_SVD_VARIABLES` interface property allows for variable substitution inside the **.svd** file. You can dynamically modify the character data of the **.svd** file by using the CMSIS_SVD_VARIABLES property.

For example, if you set the `CMSIS_SVD_VARIABLES` as shown in Example 7-4 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.

Example 7-4: Setting the CMSIS_SVD_VARIBLES Interface Property

```
set_interface_property <interface name> \
CMSIS_SVD_VARIABLES "{width} {23}"
```

Related Information

- Component Interface Tcl Reference
- CMSIS - Cortex Microcontroller Software

Specifying Component Parameters

Components can include parameterized HDL, which allows users of the component flexibility in meeting their system requirements. For example, a component might have a configurable memory size or data width, where one HDL implementation can be used in many different systems, each with unique parameters values.

The **Parameters** tab in the Component Editor allows you specify the parameters that are used to configure instances of the component in a Qsys system. You can specify various properties for each parameter that describe how the parameter is displayed and used. You can also specify a range of allowed values that are checked during the Validation phase. The **Parameters** table displays the HDL parameters that are declared in the top-level HDL module. If you have not yet created the top-level HDL file, the parameters that you create on the **Parameters** tab are included in the top-level synthesis file template created from the **Files** tab.

When the component includes HDL files, the parameters match those defined in the top-level module, and you cannot be add or remove them on the **Parameters** tab. To add or remove the parameters, edit your HDL source, and then re-analyze the file.

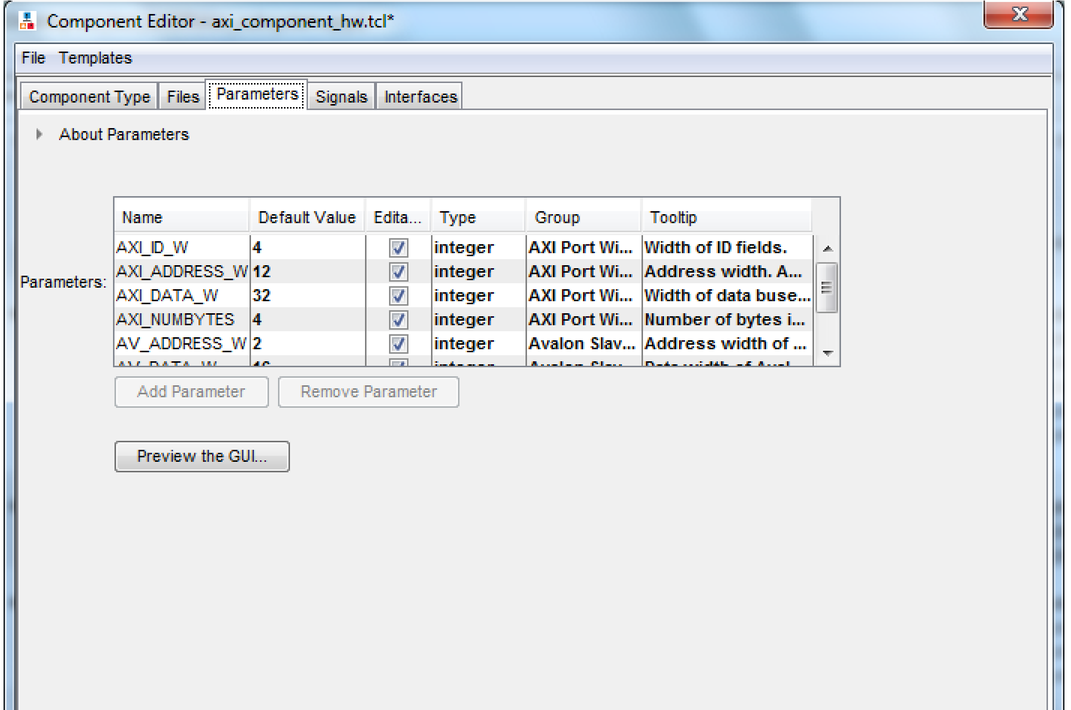
If you used the Component Editor to create a top-level template HDL file for synthesis, you can remove the newly-created file from the **Synthesis Files** list on the **Files** tab, make your parameter changes, and then re-analyze the top-level synthesis file.

You can use the **Parameters** table to specify the following information about each parameter:

- **Name**—Specifies the name of the parameter.
- **Default Value**—Sets the default value used in new instances of the component.
- **Editable**—Specifies whether or not the user can edit the parameter value.
- **Type**—Defines the parameter type as string, integer, boolean, std_logic, logic vector, natural, or positive.
- **Group**—Allows you to group parameters in parameter editor.
- **Tooltip**—Allows you to add a description of the parameter that appears when the user of the component points to the parameter in the parameter editor.

On the **Parameters** tab, you can click **Preview the GUI** at any time to see how the declared parameters appear in the parameter editor. Figure 7-4 shows parameters with their default values defined, with checks in the **Editable** column indicating that users of this component are allowed to modify the parameter value. Editable parameters cannot contain computed expressions. You can group parameters under a common heading or section in the parameter editor with the **Group** column, and a tooltip helps users of the component understand the function of the parameter. Various parameter properties allow you to customize the component's parameter editor, such as using radio buttons for parameter selections, or displaying an image.

Figure 7-4: Parameters Tab in the Components Editor



If a parameter <*n*> defines the width of a signal, the signal width must follow the format: <*n*-1>:0.

In Example 7-5, 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.

Example 7-5: _hw.tcl Created from Entries in the Parameters Tab

```
#
# 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 might not reach all AXI masters. The formula of an AXI slave ID bit width is calculated as follows:

maximum_master_id_width_in_the_interconnect + log2 (number_of_masters_in_the_same_interconnect)

For example, if an AXI slave connects to three AXI masters and the maximum AXI master ID length of the three masters is 5 bits, then the AXI slave ID is 7 bits, and is calculated as follows:

5 bits + 2 bits ($\log_2$(3 masters)) = 7

Related Information

Component Interface Tcl Reference

Allowed Ranges Parameter Property

In a component's **hw.tcl** file, you can specify valid ranges for parameters. In Qsys, validation checks each parameter value against the ALLOWED_RANGES property. If the values specified are outside of the allowed ranges, Qsys displays an error message. Specifying choices for the allowed values enables users of the component to choose the parameter value from a drop-down list or radio button in the parameter editor GUI instead of entering a value.

The ALLOWED_RANGES property is a list of valid ranges, where each range is a single value, or a range of values defined by a start and end value. Table 7-2 shows examples of the ALLOWED_RANGES property.

Table 7-2: ALLOWED_RANGES Property

ALLOWED_RANGES	Meaning
`{a b c}`	a, b, or c
`{"No Control" "Single Control" "Dual Controls"}`	Unique string values. Quotation marks are required if the strings include spaces
`{1 2 4 8 16}`	1, 2, 4, 8, or 16
`{1:3}`	1 through 3, inclusive
`{1 2 3 7:10}`	1, 2, 3, or 7 through 10 inclusive

For GUI and code example for the `ALLOWED_RANGES` property, refer to *Declaring Parameters with Custom hw.tcl Commands*.

Related Information

Declaring Parameters with Custom hw.tcl Commands on page 7-18

Types of Parameters

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

Related Information

Declaring Parameters with Custom hw.tcl Commands on page 7-18

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*.

System Information Parameters

A `SYSTEM_INFO` parameter is a parameter whose value is set automatically by the Qsys system. When you define a `SYSTEM_INFO` parameter, you provide an `information type`, and additional arguments.

For example, you can configure a parameter to store the clock frequency driving a clock input for your component. To do this, define the parameter as `SYSTEM_INFO` of type `CLOCK_RATE`:

```
set_parameter_property <param> SYSTEM_INFO CLOCK_RATE
```

You then set the name of the clock interface as the `SYSTEM_INFO` argument:

```
set_parameter_property <param> SYSTEM_INFO_ARG <clkname>
```

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.

For GUI and code example of derived parameters, refer to *Declaring Parameters with Custom hw.tcl Commands*.

Parameterized Parameter Widths

Qsys allows a `std_logic_vector` parameter to have a width that is defined by another parameter, similar to derived parameters. The width can be a constant or the name of another parameter.

Declaring 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 7-6 illustrates parameter declaration statements and includes a parameter whose value is derived during the Elaboration phase based on another parameter, instead of being assigned to a specific value. `AXI_NUMBYTES` describes the number of bytes in a word of data. Qsys calculates the `AXI_NUMBYTES` parameter from the `DATA_WIDTH` parameter by dividing by 8. The **_hw.tcl** code defines the `AXI_NUMBYTES` parameter as a derived parameter, since its value is calculated in an elaboration callback procedure.

The `AXI_NUMBYTES` parameter value is not editable, because its value is based on another parameter value.

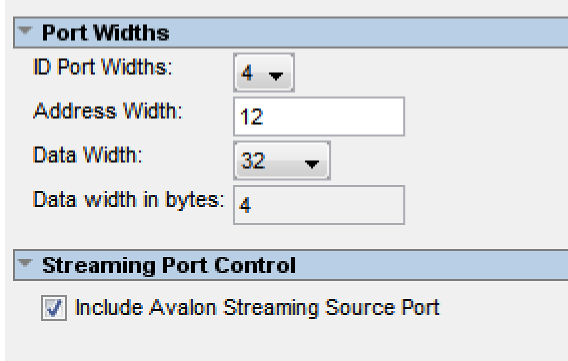
Example 7-6: Parameter Declaration Statements

```
add_parameter AXI_ADDRESS_W INTEGER 12

set_parameter_property AXI_ADDRESS_W DISPLAY_NAME \
"AXI Slave Address Width"

set_parameter_property AXI_ADDRESS_W DESCRIPTION \
"Address width."

set_parameter_property AXI_ADDRESS_W UNITS bits
set_parameter_property AXI_ADDRESS_W ALLOWED_RANGES 4:16
set_parameter_property AXI_ADDRESS_W HDL_PARAMETER true

set_parameter_property AXI_ADDRESS_W GROUP \
"AXI Port Widths"

add_parameter AXI_DATA_W INTEGER 32
set_parameter_property AXI_DATA_W DISPLAY_NAME "Data Width"
```

```
set_parameter_property AXI_DATA_W DESCRIPTION \
"Width of data buses."

set_parameter_property AXI_DATA_W UNITS bits

set_parameter_property AXI_DATA_W ALLOWED_RANGES \
{8 16 32 64 128 256 512 1024}

set_parameter_property AXI_DATA_W HDL_PARAMETER true
set_parameter_property AXI_DATA_W GROUP "AXI Port Widths"

add_parameter AXI_NUMBYTES INTEGER 4
set_parameter_property AXI_NUMBYTES DERIVED true

set_parameter_property AXI_NUMBYTES DISPLAY_NAME \
"Data Width in bytes; Data Width/8"

set_parameter_property AXI_NUMBYTES DESCRIPTION \
"Number of bytes in one word"

set_parameter_property AXI_NUMBYTES UNITS bytes
set_parameter_property AXI_NUMBYTES HDL_PARAMETER true
set_parameter_property AXI_NUMBYTES GROUP "AXI Port Widths"

add_parameter ENABLE_STREAM_OUTPUT BOOLEAN true

set_parameter_property ENABLE_STREAM_OUTPUT DISPLAY_NAME \
"Include Avalon Streaming Source Port"

set_parameter_property ENABLE_STREAM_OUTPUT DESCRIPTION \
"Include optional Avalon-ST source (default),\
or hide the interface"

set_parameter_property ENABLE_STREAM_OUTPUT GROUP \
"Streaming Port Control"

...
```

Figure 7-5 shows the parameter editor GUI generated from **Example 7-6**.

Figure 7-5: Parameter Editor Illustrating Parameter Declarations



Related Information

- Component Interface Tcl Reference
- Controlling Interfaces Dynamically with an Elaboration Callback on page 7-26

Validating Parameter Values with a Validation Callback

You can use a validation callback procedure to validate parameter values with more complex validation operations than the ALLOWED_RANGES property allows. You define a validation callback by setting the VALIDATION_CALLBACK module property to the name of the Tcl callback procedure that runs during the validation phase. In the validation callback procedure, the current parameter values is queried, and warnings or errors are reported about the component's configuration.

In Example 7-7, 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.

Example 7-7: Demo AXI Memory Example

```
set_module_property VALIDATION_CALLBACK validate
proc validate {} {
if {
  [get_parameter_value ENABLE_STREAM_OUTPUT ] &&
  ([get_parameter_value AXI_ADDRESS_W] >
  [get_parameter_value AV_DATA_W])
}
send_message error "If the optional Avalon streaming port\
is enabled, the AXI Data Width must be equal to or greater\
than the Avalon control port Address Width"
}
}
```

Related Information

- Component Interface Tcl Reference

- **Demo AXI Memory Example**

Specifying Interface and Signal Types

The **Signals** tab in the Components Editor allows you to specify the interface and signal type of each signal in the component. When you add HDL files to the **Synthesis Files** table on the **Files** tab, and then click **Analyze Synthesis Files**, the signals on the top-level module appear on the **Signals** tab.

If you have not yet created your top-level HDL file, you can click **Add Signal** to specify each top-level signal in the component. For each signal that you add, you must provide the appropriate values in the **Name**, **Interface**, **Signal Type**, **Width**, and **Direction** columns. You can use the error and warning messages at the bottom of the window to guide your selections. You can edit the signal name by double-clicking the **Name** column, and then typing the new name.

After you have analyzed 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** tab. To change the signals, edit your HDL source, and then click **Generate Synthesis File from Signals**.

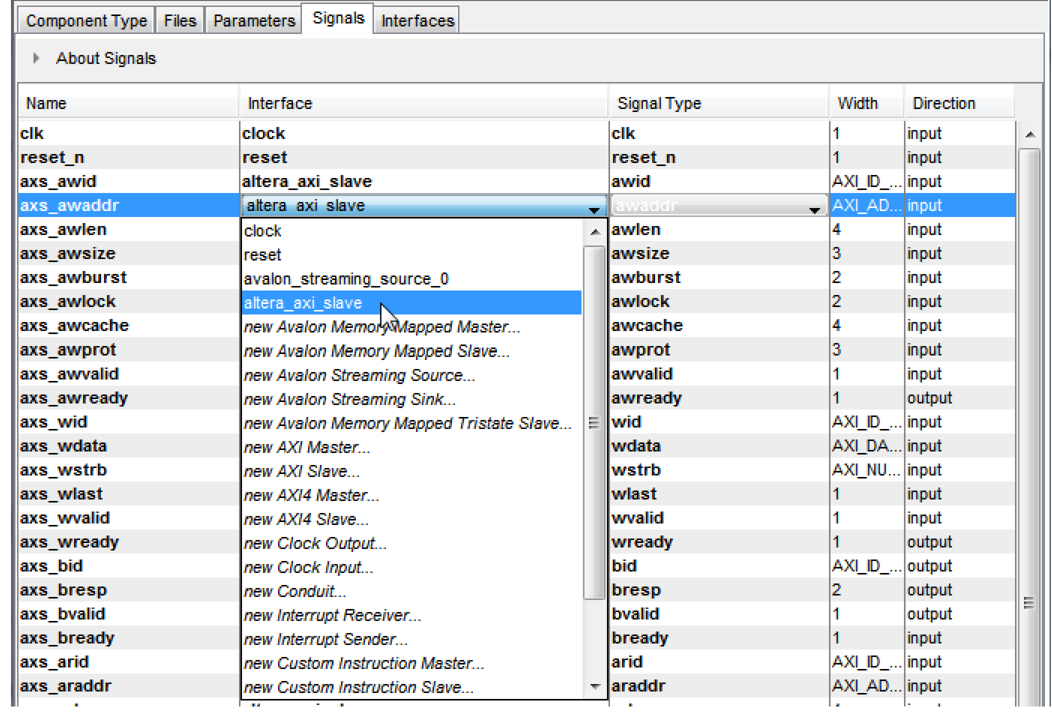
If you used the Component Editor to create a top-level template HDL file for synthesis, you can remove the newly-created file from the **Synthesis Files** list on the **Files** tab, make your signal changes, and then re-analyze the top-level synthesis file.

The **Interface** column allows you assign a signal to an interface. Each signal must belong to an interface and be assigned a legal signal type for that interface. To create a new interface of a specific type, select **new** <*interface type*> from the list; this new interface then become available in the list for subsequent signal assignments. You can highlight all of the signals in an interface and then select an Interface from the list to apply the Interface name to each signal in the interface.

You edit the interface name on the **Interface** tab; you cannot edit the interface name on the **Signals** tab.

Figure 7-6 shows the **altera_axi_slave** selection available for the **axs_awaddr** signal. Example 7-8 in the *Adding Interfaces and Managing Interface Settings* section shows the **_hw.tcl** that Qsys generates from these entries along with other interface information.

Figure 7-6: Signals Tab in the Components Editor



Related Information

Component Interface Tcl Reference

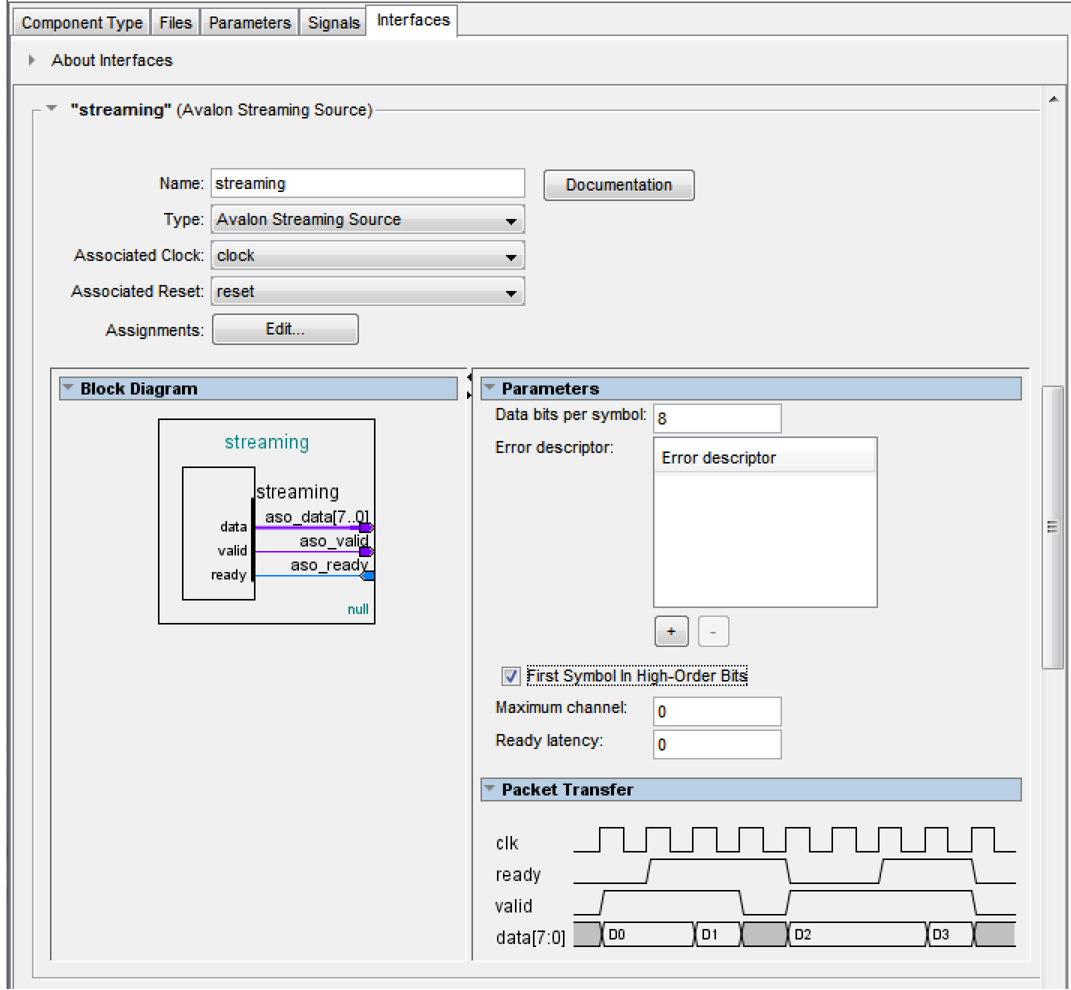
Adding Interfaces and Managing Interface Settings

The **Interfaces** tab in the Component Editor allows you to manage settings for each interface of the component. The interface name appears on the **Signals** tab, and in the Qsys **System Contents** tab when the component is added to a system.

You can configure the type and properties of each interface. Some interfaces display waveforms that illustrate the timing for the interface. If you update timing parameters, the waveforms automatically update.

You add additional interfaces by clicking **Add Interface**, and then you must specify the signals for the added interface on the **Signals** tab. You can remove interfaces that have no assigned signals by clicking **Remove Interfaces With No Signals**. Figure 7-7 shows the **Avalon Streaming Source** interface, named **streaming**.

Figure 7-7: Interfaces Tab in the Components Editor



In Example 7-8, each interface is created with the `add_interface` command. You specify the properties of each interface with the `set_interface_property` command. The interface's signals are specified with the `add_interface_port` command.

Example 7-8: _hw.tcl Created from Entries in the Interface Tab

```
#
# 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
```

Qsys refers to AXI interface parameters to build AXI interconnect. If these parameter settings are incompatible with the component's HDL behavior, Qsys interconnect and transactions might not work correctly. To prevent unexpected interconnect behavior, you must set the AXI component parameters described in Table 7-3.

Table 7-3: AXI Master and Slave Parameters

AXI Master Parameters	AXI Slave Parameters
readIssuingCapability	readAcceptanceCapability

AXI Master Parameters	AXI Slave Parameters
writeIssuingCapability	writeAcceptanceCapability
combinedIssuingCapability	combinedAcceptanceCapability
	readDataReorderingDepth

Related Information

Component Interface Tcl Reference

Creating Custom _hw.tcl Interface Settings and Properties

Example 7-9 shows clock, reset, AXI slave, and Avalon streaming interfaces using variables for the interface names that make the file easier to read and update. The interface declaration statement includes the name, type, and direction of the interface, as well as the associated clock and reset interfaces. Also in the example below, some of the AXI memory signals use parameters to specify their width.

Example 7-9: Clock, Reset, AXI Slave, and Avalon Streaming Interfaces Using Variables

```
set CLOCK_INTERFACE  "clk"
add_interface $CLOCK_INTERFACE clock end
add_interface_port $CLOCK_INTERFACE clk clk Input 1

set RESET_INTERFACE  "reset"
add_interface $RESET_INTERFACE  reset end
set_interface_property $RESET_INTERFACE  associatedClock clk
set_interface_property $RESET_INTERFACE  synchronousEdges DEASSERT
add_interface_port reset reset_n reset_n Input 1

set SLAVE_INTERFACE  "slave"
add_interface $SLAVE_INTERFACE axi end
set_interface_property $SLAVE_INTERFACE associatedClock "clk"
set_interface_property $SLAVE_INTERFACE associatedReset "reset"

set_interface_property $SLAVE_INTERFACE \
readAcceptanceCapability 1
...
add_interface_port $SLAVE_INTERFACE axs_wdata wdata \
Input AXI_DATA_W

add_interface_port $SLAVE_INTERFACE axs_wstrb wstrb \
Input AXI_NUMBYTES

add_interface_port $SLAVE_INTERFACE axs_wlast wlast Input 1
...
set STREAMING_INTERFACE "streaming"
add_interface $STREAMING_INTERFACE avalon_streaming start
set_interface_property $STREAMING_INTERFACE associatedClock "clk"
...
add_interface_port $STREAMING_INTERFACE aso_data data Output 8
```

```
add_interface_port $STREAMING_INTERFACE aso_valid valid Output 1
add_interface_port $STREAMING_INTERFACE aso_ready ready Input 1
```

Related Information

Component Interface Tcl Reference

Controlling Interfaces Dynamically with an Elaboration Callback

You can allow user parameters to dynamically control your component's behavior with a an elaboration callback procedure during the elaboration phase. Using an elaboration callback allows you to change interface properties, remove interfaces, or add new interfaces as a function of parameter values. 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 7-10 shows an Avalon-ST source interface that is optionally included in an instance of the component, based on the ENABLE_STREAM_OUTPUT parameter. The `ENABLE_STREAM_OUTPUT` parameter was defined previously in the `module_property VALIDATION_CALLBACK`, and the streaming interface was defined previously in the static portion of the HDL file.

Example 7-10: Optional Avalon-ST Source Interface Specified with an Elaboration Callback

```
set_module_property ELABORATION_CALLBACK elaborate

proc elaborate {} {

   # Optionally disable the Avalon- ST data output

    if{[ get_parameter_value ENABLE_STREAM_OUTPUT] == "false" }{

       set_port_property aso_data    termination true
       set_port_property aso_valid   termination true
       set_port_property aso_ready   termination true
       set_port_property aso_ready   termination_value 0
    }
  # Calculate the Data Bus Width in bytes

    set bytewidth_var [expr [get_parameter_value AXI_DATA_W]/8]
    set_parameter_value AXI_NUMBYTES $bytewidth_var
}
```

Related Information

- Component Interface Tcl Reference
- Creating Custom _hw.tcl Interface Settings and Properties on page 7-25
- Validating Parameter Values with a Validation Callback on page 7-20

Controlling 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 7-11 shows a fileset callback using parameters to control filesets in two different ways. The `RAM_VERSION` parameter chooses between two different source files to control the implementation of a RAM block. For the top-level source file, a custom Tcl routine generates HDL that optionally includes control and status registers, depending on the value of the `CSR_ENABLED` parameter.

During the generation phase, Qsys creates a a top-level Qsys system HDL wrapper module to instantiate the component top-level module, and applies the component's parameters, for any parameter whose parameter property `HDL_PARAMETER` is set to true.

Example 7-11: Fileset Callback Using Parameters to Control Filesets

```
#Create synthesis fileset with fileset_callback and set top level

add_fileset my_synthesis_fileset QUARTUS_SYNTH fileset_callback

set_fileset_property my_synthesis_fileset TOP_LEVEL \
demo_axi_memory

# Create Verilog simulation fileset with same fileset_callback
# and set top level

add_fileset my_verilog_sim_fileset SIM_VERILOG fileset_callback

set_fileset_property my_verilog_sim_fileset TOP_LEVEL \
demo_axi_memory

# Add extra file needed for simulation only

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH \
verification_lib/verbosity_pkg.sv

# Create VHDL simulation fileset (with Verilog files
# for mixed-language VHDL simulation)

add_fileset my_vhdl_sim_fileset SIM_VHDL fileset_callback
set_fileset_property my_vhdl_sim_fileset TOP_LEVEL demo_axi_memory

add_fileset_file verbosity_pkg.sv SYSTEM_VERILOG PATH
verification_lib/verbosity_pkg.sv
```

```
# Define parameters required for fileset_callback

add_parameter RAM_VERSION INTEGER 1
set_parameter_property RAM_VERSION ALLOWED_RANGES {1 2}
set_parameter_property RAM_VERSION HDL_PARAMETER false
add_parameter CSR_ENABLED BOOLEAN enable
set_parameter_property CSR_ENABLED HDL_PARAMETER false

# Create Tcl callback procedure to add appropriate files to
# filesets based on parameters

proc fileset_callback { entityName } {
  send_message INFO "Generating top-level entity $entityName"
  set ram [get_parameter_value RAM_VERSION]
  set csr_enabled [get_parameter_value CSR_ENABLED]

  send_message INFO "Generating memory
  implementation based on RAM_VERSION $ram "

   if {$ram == 1} {
    add_fileset_file single_clk_ram1.v VERILOG PATH \
    single_clk_ram1.v
   } else  {
    add_fileset_file single_clk_ram2.v VERILOG PATH \
    single_clk_ram2.v
 }

send_message INFO "Generating top-level file for \
CSR_ENABLED $csr_enabled"

generate_my_custom_hdl $csr_enabled demo_axi_memory_gen.sv

add_fileset_file demo_axi_memory_gen.sv VERILOG PATH \
demo_axi_memory_gen.sv
}
```

Related Information

- **Component Interface Tcl Reference**
- **Specifying Files for Synthesis and Simulation** on page 7-9

Creating a Composed Component or Subsystem

A composed component is a subsystem containing instances of other components. Unlike an HDL-based component, a composed component's HDL is created by generating HDL for the components in the subsystem, in addition to the Qsys interconnect to connect the subsystem instances.

You can add child instances in a composition callback of the **_hw.tcl** file.

With a composition callback, you can also instantiate and parameterize subcomponents 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 subcomponents 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.

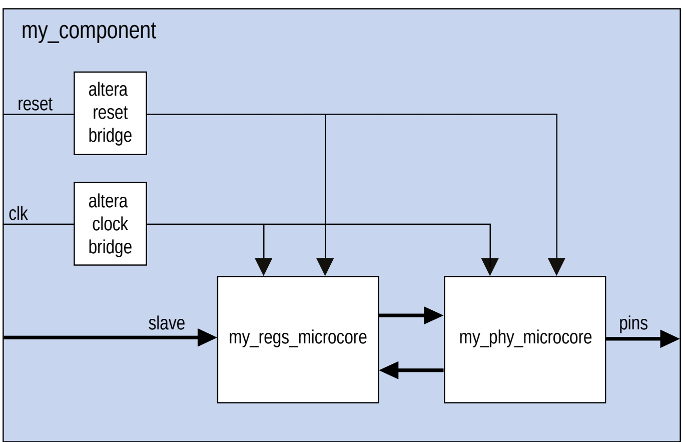
Exporting an interface means that you are making the interface visible from the outside of your component, instead of connecting it internally. You can set the `EXPORT_OF` property of the externally visible interface from the main program or the composition callback, to indicate that it is an exported 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 7-8 shows a block diagram for the composed component in **Example 7-12**.

Figure 7-8: Top-Level of a Composed Component



In **Example 7-12**, Qsys connects the components, and also connects the clocks and resets. Note that clock and reset bridge components are required to allow both subcomponents to see common clock and reset inputs.

Example 7-12: Composed _hw.tcl File that Instantiates two Subcomponents

```
package require -exact qsys 13.1
set_module_property name my_component
set_module_property COMPOSITION_CALLBACK composed_component

proc composed_component {} {
  add_instance clk altera_clock_bridge
  add_instance reset altera_reset_bridge
  add_instance regs my_regs_microcore
  add_instance phy my_phy_microcore

  add_interface clk clock end
  add_interface reset reset end
  add_interface slave avalon slave
  add_interface pins conduit end

  set_interface_property clk EXPORT_OF clk.in_clk
  set_instance_property_value reset synchronous_edges deassert
  set_interface_property reset EXPORT_OF reset.in_reset
  set_interface_property slave EXPORT_OF regs.slave
  set_interface_property pins  EXPORT_OF phy.pins

  add_connection clk.out_clk reset.clk
  add_connection clk.out_clk regs.clk
  add_connection clk.out_clk phy.clk
  add_connection reset.out_reset regs.reset
  add_connection reset.out_reset phy.clk_reset
  add_connection regs.output phy.input
  add_connection phy.output regs.input
}
```

Related Information

Component Interface Tcl Reference

Creating a Component With Differing Structural Qsys View and Generated Output Files

There are cases where it might be beneficial to have the structural Qsys system view of a component differ from the generated synthesis output files. The structural composition callback allows you to define a structural hierarchy for a component separately from the generated output files.

One application of this feature is for IP designers who want to send out a placed-and-routed component that represents a Qsys system in order to ensure timing closure for the end-user. In this case, the designer creates a design partition for the Qsys system, and then exports a post-fit Quartus II Exported Partition File (**.qxp**) when satisfied with the placement and routing results.

The designer specifies a **.qxp** file as the generated synthesis output file for the new component. The designer can specify whether to use a simulation output fileset for the custom simulation model file, or to use simulation output files generated from the original Qsys system.

When the end-user adds this component to their Qsys system, the designer wants the end-user to see a structural representation of the component, including lower-level components and the address map of the original Qsys system. This structural view is a logical representation of the component that is used during the elaboration and validation phases in Qsys.

To specify a structural representation of the component for Qsys, the designer connects components, or generates a hardware Tcl description of the Qsys system, and then insert the Tcl commands into a structural composition callback. Example 7-13 shows an **_hw.tcl** file with a structural composition callback and a **.qxp** file as the generated output file. To invoke the structural composition callback use the command:

```
set_module_property STRUCTURAL_COMPOSITION_CALLBACK structural_hierarchy
```

Example 7-13: Structural Composition Callback

```
package require -exact qsys 13.1
set_module_property name example_structural_composition

set_module_property STRUCTURAL_COMPOSITION_CALLBACK \
structural_hierarchy

add_fileset synthesis_fileset QUARTUS_SYNTH \
synth_callback_procedure

add_fileset simulation_fileset SIM_VERILOG \
sim_callback_procedure

set_fileset_property synthesis_fileset TOP_LEVEL \
my_custom_component

set_fileset_property simulation_fileset TOP_LEVEL \
my_custom_component

proc structural_hierarchy {} {

# called during elaboration and validation phase
# exported ports should be same in structural_hierarchy
# and generated QXP

# These commands could come from the exported hardware Tcl

 add_interface clk clock sink
 add_interface reset reset sink

 add_instance clk_0 clock_source
 set_interface_property clk EXPORT_OF clk_0.clk_in
 set_interface_property reset EXPORT_OF clk_0.clk_in_reset

 add_instance pll_0 altera_pll
 # connections and connection parameters
 add_connection clk_0.clk pll_0.refclk clock
 add_connection clk_0.clk_reset pll_0.reset reset
}
```

```
proc synth_callback_procedure { entity_name } {

# the QXP should have the same name for ports
# as exportedin structural_hierarchy

 add_fileset_file my_custom_component.qxp QXP PATH \
 "my_custom_component.qxp"
}

proc sim_callback_procedure { entity_name } {

# the simulation files should have the same name for ports as
# exported in structural_hierarchy

add_fileset_file my_custom_component.v VERILOG PATH \
"my_custom_component.v"
 ….
 ….
}
```

Related Information

Creating a Composed Component or Subsystem on page 7-28

Adding 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.

Note: You cannot add child instances in a static part of a **_hw.tcl** because for Qsys 13.1, the `add_hdl_instance` and `set_instance_parameter_value` commands are not supported in global context.

With an elaboration callback, you can also instantiate and parameterize subcomponents with the `add_hdl_instance` command as a function of the parent component's parameter values.

When you instantiate multiple nested components, you must create a unique variation name for each component with the `add_hdl_instance` command. Prefixing a variation name with the parent component name prevents conflicts in a system. The variation name can be the same across multiple parent components if the generated parameterization of the nested component is exactly the same.

Note: If you do not adhere to the above naming variation guidelines, Qsys validation-time errors occur, which are often difficult to debug.

Related Information

- Static Components on page 7-33
- Generated Components on page 7-34

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 same HDL regardless of parameterization, Qsys generates static IPs only once across multiple instantiations, meaning they have the same top-level name set. Example 7-14 shows typical usage of the add_hdl_instance command for static components.

Example 7-14: add_hdl_instance for Static Components

```
package require -exact qsys 13.1

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 7-14 generates a wrapper file for the instance name specified in the **_hw.tcl** file. Example 7-15 shows the top-level HDL instance and the wrapper file created by Qsys.

Example 7-15: Top-Level HDL Instance and Wrapper File

```
//Top Level Component HDL
module basic_static (input_wire, output_wire, inout_wire);
input [31:0] input_wire;
output [31:0] output_wire;
inout [31:0] inout_wire;

// Instantiation of the instance added via add_hdl_instance
// command. This is an example of how the instance added via
// the add_hdl_instance command can be used
// in the top-level file of the component.

emif_instance_name fixed_name_instantiation_in_top_level(
.pll_ref_clk (input_wire), // pll_ref_clk.clk
.global_reset_n (input_wire), // global_reset.reset_n
```

```
.soft_reset_n (input_wire), // soft_reset.reset_n
...
... );
endmodule

//Wrapper for added HDL instance
// emif_instance_name.v
// Generated using ACDS version 13.1

`timescale 1 ps / 1 ps
module emif_instance_name (
input wire pll_ref_clk, // pll_ref_clk.clk
input wire global_reset_n, // global_reset.reset_n
input wire soft_reset_n, // soft_reset.reset_n
output wire afi_clk, // afi_clk.clk
...
...);
example_addhdlinstance_system
_add_hdl_instance_example_0_emif_instance
_name_emif_instance_name emif_instance_name (

.pll_ref_clk (pll_ref_clk), // pll_ref_clk.clk
.global_reset_n (global_reset_n), // global_reset.reset_n
.soft_reset_n (soft_reset_n), // soft_reset.reset_n
...
...);
endmodule
```

Generated Components

A generated component's fileset callback allows an instance of the component to create unique HDL design files based on the instance's parameter values. For example, you can write a fileset callback to include a control and status interface based on the value of a parameter. The callback overcomes a limitation of HDL languages, which do not allow runtime parameters.

Generated components change their generation output (HDL) based on their parameterization. If a component is generated, then any component that might instantiate it with multiple parameter sets must also be considered generated, since its HDL changes with its parameterization. This case has an effect that propagates up to the top-level of a design.

Since generated components are generated for each unique parameterized instantiation, when implementing the `add_hdl_instance` command, you cannot use the same fixed name (specified using `instance_name`) for the different variants of the child HDL instances. To facilitate unique naming for the wrapper of each unique parameterized instantiation of child HDL instances, you must use the following command so that Qsys generates a unique name for each wrapper. You can then access this unique wrapper name with a fileset callback so that the instances are instantiated inside the component's top-level HDL.

- To declare auto-generated fixed names for wrappers, use the command:

  ```
  set_instance_property instance_name HDLINSTANCE_USE_GENERATED_NAME \
  true
  ```

  You can only use this command with a generated component, and is used in the global context, or in an elaboration callback

- To obtain auto-generated fixed name with a fileset callback, use the command:

  ```
  get_instance_property instance_name HDLINSTANCE_GET_GENERATED_NAME
  ```

  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 7-16 shows typical usage of the `add_hdl_instance` command for generated components.

Example 7-16: add_hdl_instance for Generated Components

```
package require -exact qsys 13.1
set_module_property name generated_toplevel_component
set_module_property ELABORATION_CALLBACK elaborate
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH generate
add_fileset SIM_VERILOG SIM_VERILOG generate
add_fileset SIM_VHDL SIM_VHDL generate

proc elaborate {} {

 # Actual API to instantiate an IP Core
 add_hdl_instance emif_instance_name altera_mem_if_ddr3_emif

 # Make sure the parameters are set appropriately
 set_instance_parameter_value emif_instance_name SPEED_GRADE {7}

 ...
 # instruct Qsys to use auto generated fixed name
 set_instance_property emif_instance_name \
 HDLINSTANCE_USE_GENERATED_NAME 1
}

proc generate { entity_name } {

 # get the autogenerated name for emif_instance_name added
 # via add_hdl_instance

 set autogeneratedfixedname [get_instance_property \
 emif_instance_name HDLINSTANCE_GET_GENERATED_NAME]

 set fileID [open "generated_toplevel_component.v" r]
 set temp ""

 # read the contents of the file
```

```
 while {[eof $fileID] != 1} {
 gets $fileID lineInfo

 # replace the top level entity name with the name provided
 # during generation

 regsub -all "substitute_entity_name_here" $lineInfo \
 "${entity_name}" lineInfo

 # replace the autogenerated name for emif_instance_name added
 # via add_hdl_instance

 regsub -all "substitute_autogenerated_emifinstancename_here" \
 $lineInfo"${autogeneratedfixedname}" lineInfo \
 append temp "${lineInfo}\n"
}

# adding a top level component file

add_fileset_file ${entity_name}.v VERILOG TEXT $temp
}
```

Example 7-16 generates a wrapper file for the instance name specified in the **_hw.tcl** file. Example 7-17 shows the top-level HDL instance and the wrapper file created by Qsys

Example 7-17: Top-Level HDL Instance and Wrapper File

```
// Top Level Component HDL

module substitute_entity_name_here (input_wire, output_wire,
inout_wire);

input [31:0] input_wire;
output [31:0] output_wire;
inout [31:0] inout_wire;

// Instantiation of the instance added via add_hdl_instance
// command. This is an example of how the instance added
// via add_hdl_instance command can be used
// in the top-level file of the component.

substitute_autogenerated_emifinstancename_here
fixed_name_instantiation_in_top_level (
.pll_ref_clk (input_wire), // pll_ref_clk.clk
.global_reset_n (input_wire), // global_reset.reset_n
.soft_reset_n (input_wire), // soft_reset.reset_n
...
... );
endmodule
```

```
// Wrapper for added HDL instance
// generated_toplevel_component_0_emif_instance_name.v is the
// auto generated //emif_instance_name
// Generated using ACDS version 13.

`timescale 1 ps / 1 ps
module generated_toplevel_component_0_emif_instance_name (
input wire pll_ref_clk, // pll_ref_clk.clk
input wire global_reset_n, // global_reset.reset_n
input wire soft_reset_n, // soft_reset.reset_n
output wire afi_clk, // afi_clk.clk
...
...);
example_addhdlinstance_system_add_hdl_instance_example_0_emif
_instance_name_emif_instance_name emif_instance_name (

.pll_ref_clk (pll_ref_clk), // pll_ref_clk.clk
.global_reset_n (global_reset_n), // global_reset.reset_n
.soft_reset_n (soft_reset_n), // soft_reset.reset_n
...
...);
endmodule
```

Related Information

Controlling File Generation Dynamically with Parameters and a Fileset Callback on page 7-27

Design Guidelines for Adding Component Instances

In order to promote standard and predictable results when generating static and generated components, Altera recommends the following best-practices:

- For two different parameterizations of a component, a component must never generate a file of the same name with different instantiations. The contents of a file of the same name must be identical for every parameterization of the component.
- If a component generates a nested component, it must never instantiate two different parameterizations of the nested component using the same instance name. If the parent component's parameterization affects the parameters of the nested component, the parent component must use a unique instance name for each unique parameterization of the nested component.
- Static components that generate differently based on parameterization have the potential to cause problems in the following cases:
  - Different file names with the same entity names, results in same entity conflicts at compilation-time
  - Different contents with the same file name results in overwriting other instances of the component, and in either file, compile-time conflicts or unexpected behavior.
- Generated components that generate files not based on the output name and that have different content results in either compile-time conflicts, or unexpected behavior.

Document Revision History

Table 7-4 indicates edits made to the *Creating Qsys Components* content since its creation.

Table 7-4: Document Revision History

Date	Version	Changes
November 2013	13.1.0	• Added add_hdl_instance. • Added *Creating a Component With Differing Structural Qsys View and Generated Output Files* .
May 2013	13.0.0	• Consolidated content from other Qsys chapters. • Added *Upgrading IP Components to the Latest Version*. • Updated for AMBA APB support.
November 2012	12.1.0	• Added AMBA AXI4 support. • Added the **demo_axi_memory** example with screen shots and example **_hw.tcl** code.
June 2012	12.0.0	• Added new tab structure for the Component Editor. • Added AMBA AXI3 support.
November 2011	11.1.0	Template update.
May 2011	11.0.0	• Removed beta status. • Added Avalon Tri-state Conduit (Avalon-TC) interface type. • Added many interface templates for Nios custom instructions and Avalon-TC interfaces.
December 2010	10.1.0	Initial release.

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

Related Information

Quartus II Handbook Archive