Vivado Design Suite User Guide

Designing IP Subsystems Using IP Integrator

UG994 (v2019.1) May 22, 2019

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
Chapter 2: Creating a Block Design	Updated Constant, Constant, and Slice section
Chapter 3: Creating a Memory Map	Added Terminology section
Chapter 5: Cross-Probing Timing Paths	Added information on the Cross-Probing feature
Chapter 12: Using Third-Party Synthesis Tools in IP Integrator	Added section on downloading board files from Git Hub
Chapter 13: Referencing RTL Modules	Added compatibility information

XILINX

Table of Contents

Chapter 1

Getting Started with Vivado IP Integrator

As FPGAs become larger and more complex, and as design schedules become shorter, use of third-party IP and design reuse is becoming mandatory. Xilinx® recognizes the challenges designers face, and to aid designers with design and reuse issues, has created a powerful feature within the Vivado® Design Suite called the Vivado IP integrator.

The Vivado IP integrator lets you create complex system designs by instantiating and interconnecting IP from the Vivado IP catalog on a design canvas. You can create designs interactively through the IP integrator canvas GUI or programmatically through a Tcl programming interface. Designs are typically constructed at the interface level (for enhanced productivity) but may also be manipulated at the port level (for precision design manipulation).

An interface is a grouping of signals that share a common function. An AXI4-Lite master, for example, contains a large number of individual signals plus multiple buses, which are all required to make a connection. If each signal or bus is visible individually on an IP symbol, the symbol is visually very complex. By grouping these signals and buses into an interface, the following advantages can be realized:

- A single connection in IP integrator (or Tcl command) creates a master to slave connection.
- The graphical representation of this connection is a simple, single connection.
- Design Rule Checks (DRCs) that are aware of the specific interface can be run to assure that all the required signals are connected properly.

A key strength of IP integrator is that it provides a Tcl command language extension mechanism for its automation services so that system design tasks, such as parameter propagation, can be optimized per-IP or application domain.

Additionally, IP integrator implements dynamic, run-time DRCs to ensure that connections between the IP in an IP integrator design are compatible and that the IP themselves are properly configured.

Creating a Block Design

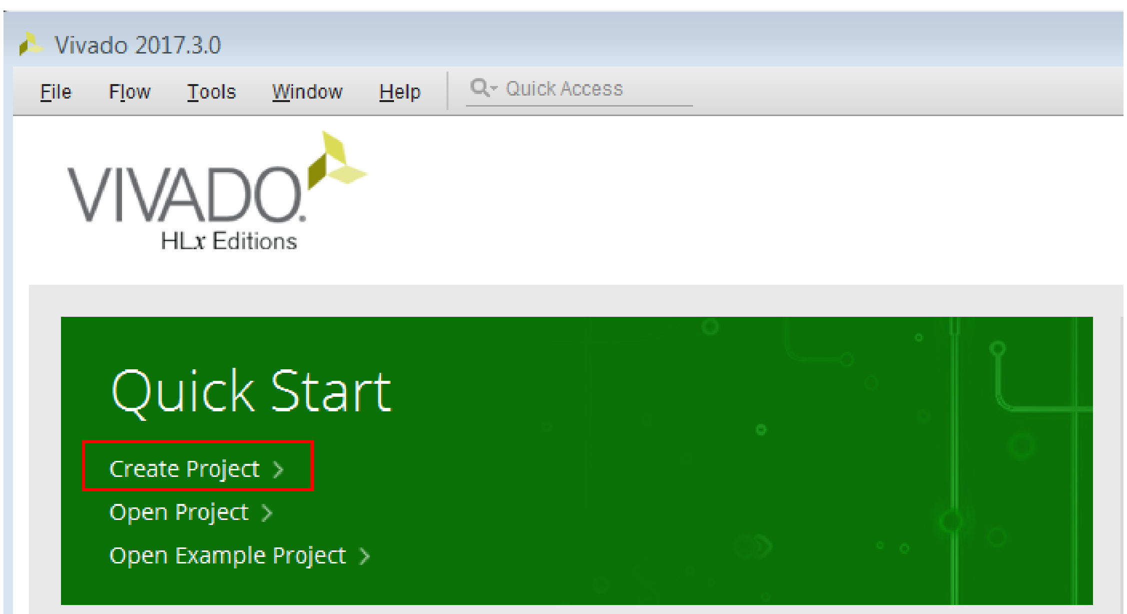
This chapter describes the basic features and functionality of Vivado® IP integrator.

Creating a Project

You can create entire designs using IP integrator; however, the typical design consists of HDL, IP, and IP integrator block designs (BDs). This section is an introduction to creating a new IP integrator-based design.

To create a project, click **Create Project** in the Vivado® IDE graphical user interface (GUI), as shown in the following figure.

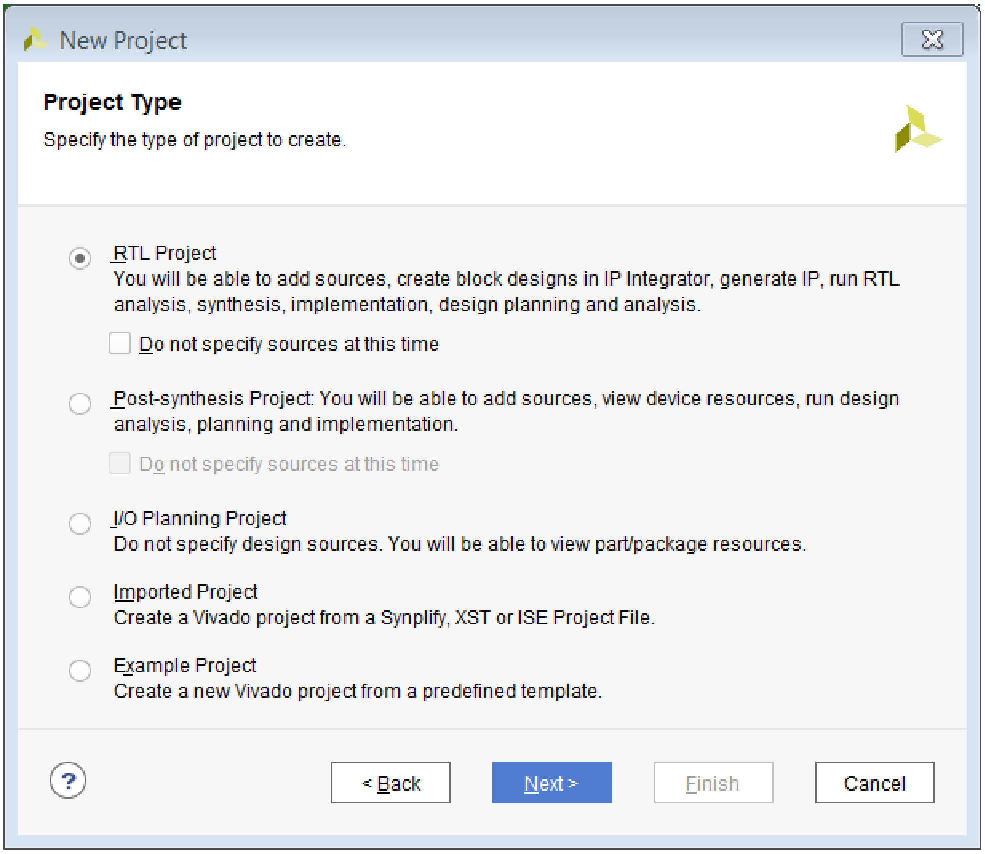
Figure 1: **Create Project**



The Vivado Design Suite supports many different types of design projects. See this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)for more information.

To add or create a BD in a project, create an RTL project, or select **Example Project**. You can add HDL design files, user constraints, and other types of design source files to the project using the New Project wizard.

Figure 2: **New Project Wizard**



After adding design sources, existing IP, and design constraints, you can also select the default Xilinx® device or platform board to target for the project, as shown in the following figure. For more information, see Chapter 11: Using the Platform Board Flow in IP Integrator.

IMPORTANT! *The Vivado tools support multiple versions of Xilinx target boards, so carefully select your target hardware.*

***Note*:** Click the blue command links to see more information about the Tcl commands in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

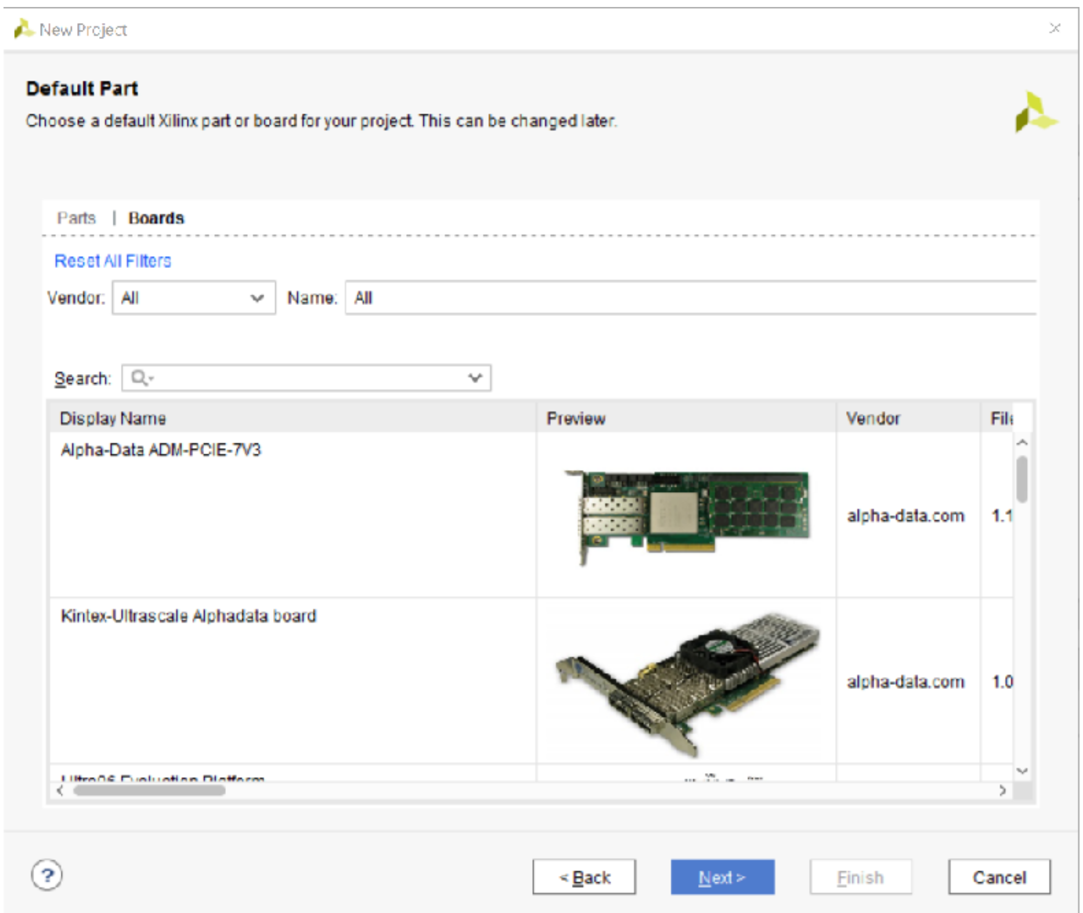
You can use the Tcl equivalent commands for creating a project, which are a combination of the create_project and set_property commands:

```
create_project <project_name> <dir_name>/xx -part xc7k325tffg900-2
set_property BOARD_PART xilinx.com:kc705:part0:1.2 [current_project]
set_property TARGET_LANGUAGE vhdl [current_project]
```

***Note*:** When displaying the Tcl commands in this document, the <> characters are used to designate variables that are specific to your design. Do not include the <> symbols in the command string.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) for information on specific Tcl commands.

Figure 3: **New Project Wizard: Default Part Page**

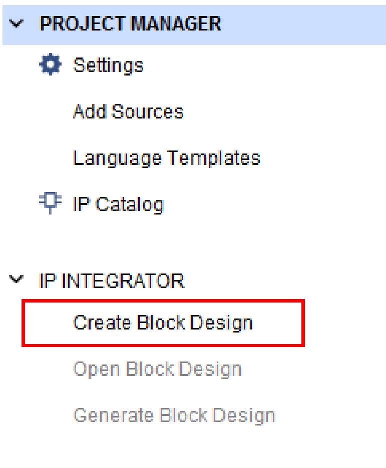


Creating a Block Design

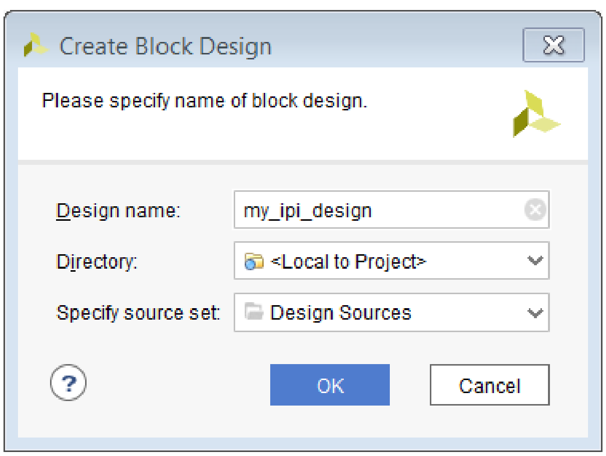
You can create a block design (BD) inside the current project directory, or outside of the project directory structure. A common use case for creating the BD outside of a project is to use the BD in non-project mode, or to use it in multiple projects, or to use it in a team-based design flow.

To create a new BD:

1. In the Flow Navigator, click **Create Block Design** under the IP INTEGRATOR heading, as shown in the following figure.



   The Create Block Design dialog box opens, as shown in the following figure.



2. Specify the Design name, Directory, and Specify source set for the design.

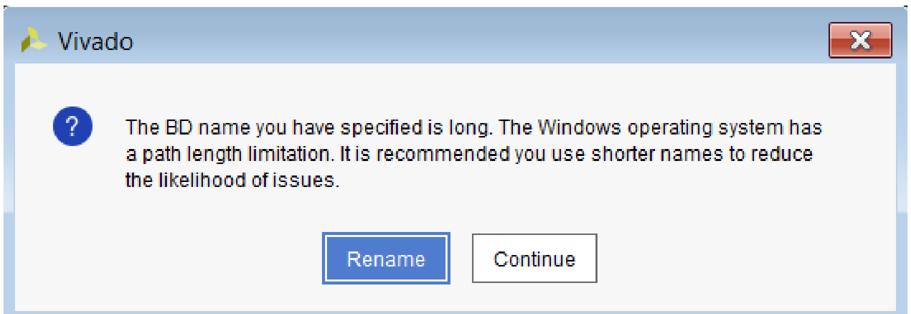
   The default value for the Directory field is <Local to Project>. To override the default value, click **Directory** and select **Choose Location**.

3. Click **OK**.

The equivalent Tcl command to create a BD is create_bd_design. The syntax of the command is as follows:

```
create_bd_design <your_design_name>
```

IMPORTANT! *Limit block design names to 25 characters or fewer to avoid any problems with the path length limitation of the Windows OS. When the specified name exceeds 25 characters, the Vivado tool issues a warning, as shown in the following figure.*



The Create Block Design creates an empty BD on disk, which is not automatically removed if the BD is closed without saving.

4. Manually delete the empty BD from the Sources window of the Vivado IDE by doing one of the following:

   - Click the **Delete** button
   - Use the remove_files Tcl command, shown as follows:

```
remove_files <project_name>/<project_name>.srcs/sources_1/bd/<bd_name>/
<bd_name>.bd

file delete -force <project_name>/<project_name>.srcs/sources_1/bd/<bd_name>
```

***Note*:** Starting in Vivado Design Suite version 2018.3, the block design schema format has changed from XML to JSON. When you open a block design that uses the older XML schema in Vivado 2018.3 or later, click Save to convert the format from XML to JSON. The following INFO message notifies you of the schema change.

```
INFO: [BD 41-2124] The block design file <block_design.bd> has changed from
an XML
format to a JSON format. All flows are expected to work as in prior
versions of
Vivado. Please contact your Xilinx Support representative, in case of any
issues.
```

IMPORTANT! *The format conversion from XML to JSON occurs only during a Save operation.*

Designing with IP Integrator

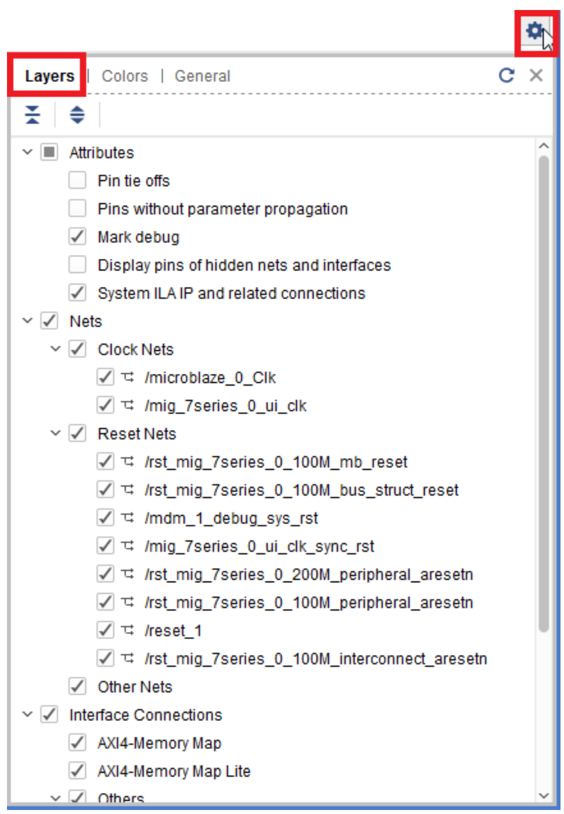
After you create the BD, the Vivado IP integrator provides a design canvas that you can use to construct your design. This canvas can be re-sized in the Vivado IDE GUI. You can double-click the design canvas tab at the upper-left corner of the diagram to increase the size of the diagram.

When you double-click the tab again, the view returns to the default layout. You can move the design canvas to a separate monitor by clicking the Float button in the upper-right corner of the diagram, and moving the window as needed.

Displaying Layers in the Block Design

To display block design (BD) layers, click the Settings Button . You can select the Attributes, Nets, and Interface connections that you want to view or hide by selecting or deselecting the associated check boxes.

Figure 4: **Viewing/Hiding Information on the IP Integrator Canvas**

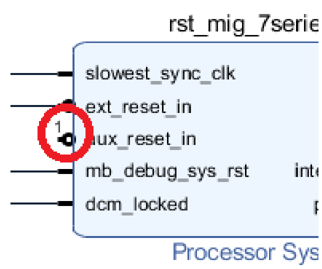


Attributes

You can display or hide several attributes of the BD by checking or un-checking the options. The following attributes can be modified.

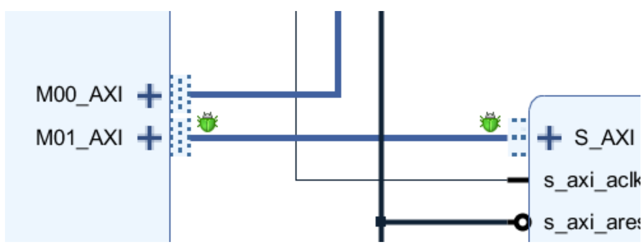
- **Pin tie offs:** Pins that have a tie-off value specified, for example, '0' or '1' can be displayed by checking the Pin tie offs option.

Figure 5: **Viewing/Hiding Pin Tie-offs on the Pins of IP Symbols**



- **Pins without parameter propagation:** Show or hide the pins that do not propagate parameters.
- **Mark Debug:** Show or hide pins that have been marked for debug. Nets marked for debug have a bug symbol placed on them.

Figure 6: **Nets Marked for Debug**



- **Display pins of hidden nets and interfaces:** Works in conjunction with the Nets or Interface Connections option. If a net has been hidden by un-checking the appropriate net, then the pins that are connected by the net also are hidden. This option displays the pins in question, even though the nets might be hidden.
- **System ILA IP and related connections:** Shows or hides the instantiation of the System ILA IP and all the connected nets. When a net is marked for debug, the designer assistance feature offers assistance to connect the net being debugged to a System ILA IP. If there are multiple System ILA IP in the BD, this could unnecessarily clutter the BD canvas. Un-checking this option hides all the System ILA IP instances and all connected nets to them.

Nets

Several types of nets such as clock nets, reset nets, data nets or simply other unclassified type of nets can be hidden or shown on the BD canvas by selecting the appropriate check box.

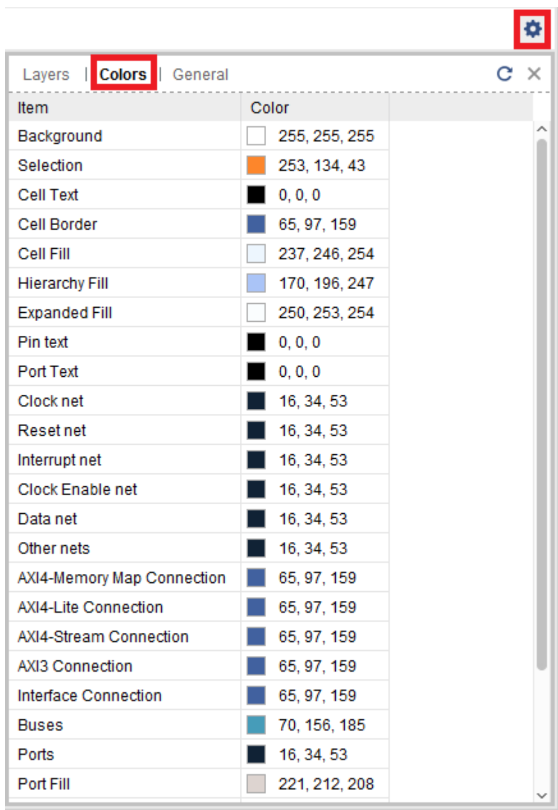
Interface Connection

Interface connections can also be shown or hidden by selecting the options under this category.

Defining Colors in the Block Design

You can change the background color of the diagram canvas and other objects from the default color. As shown in the following figure, you can click the **Block Design Options → Colors** button in the upper-left corner of the diagram to change the color.

Figure 7: **Changing the IP Integrator Background Color**



Notice that you can control the colors of almost every object displayed in an IP integrator diagram.

For example, changing the Background color to 240,240,240 as shown above makes the background light gray. To hide the options, either click the **Close** button in the upper-right corner, or click the **Settings** button again.

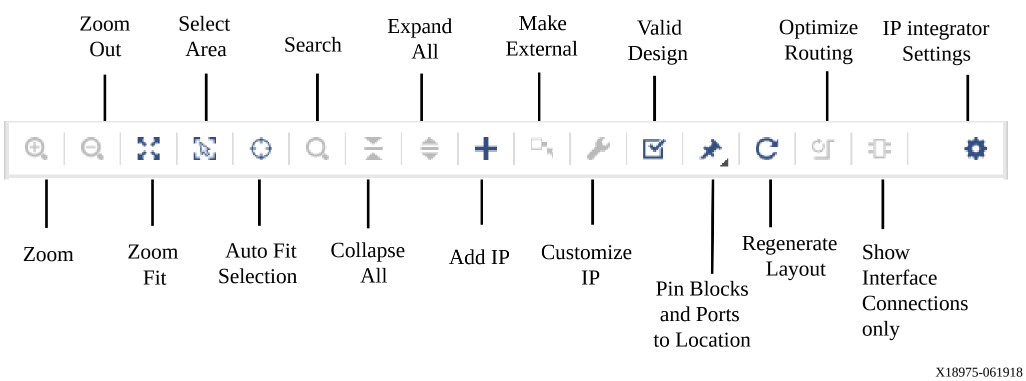
Using Mouse Strokes and the Toolbar Buttons

- **Zoom Area:** A southeast stroke (upper-left to lower-right)

- **Zoom Fit:** A northwest stroke (lower-right to upper-left)
- **Zoom In:** A southwest stroke (upper-right to lower-left)
- **Zoom Out:** A northeast stroke (lower-left to upper-right)

The toolbar buttons on the top side of the design canvas invoke the commands shown in the following figure:

Figure 8: **IP Integrator Action Buttons**



Adding Comments to Block Designs

You can add comments anywhere in the BD.

1. Right-click anywhere in the BD and select **Create Comment**.

   This creates a comment box where you can type comments:

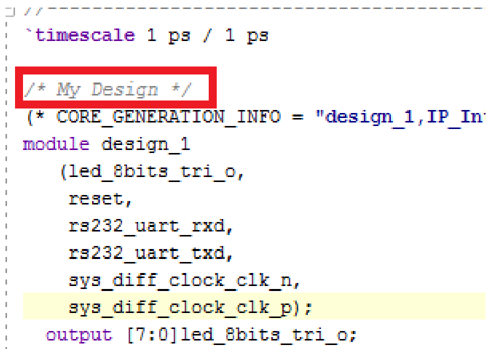
   

   The corresponding Tcl commands are as follows:

```
            set_property USER_COMMENTS.comment_0 {}  [current_bd_design]
            set_property USER_COMMENTS.comment_0 {Enter Comments here}
[current_bd_design]
            set_property USER_COMMENTS.comment_0 {My Design}
[current_bd_design]
```

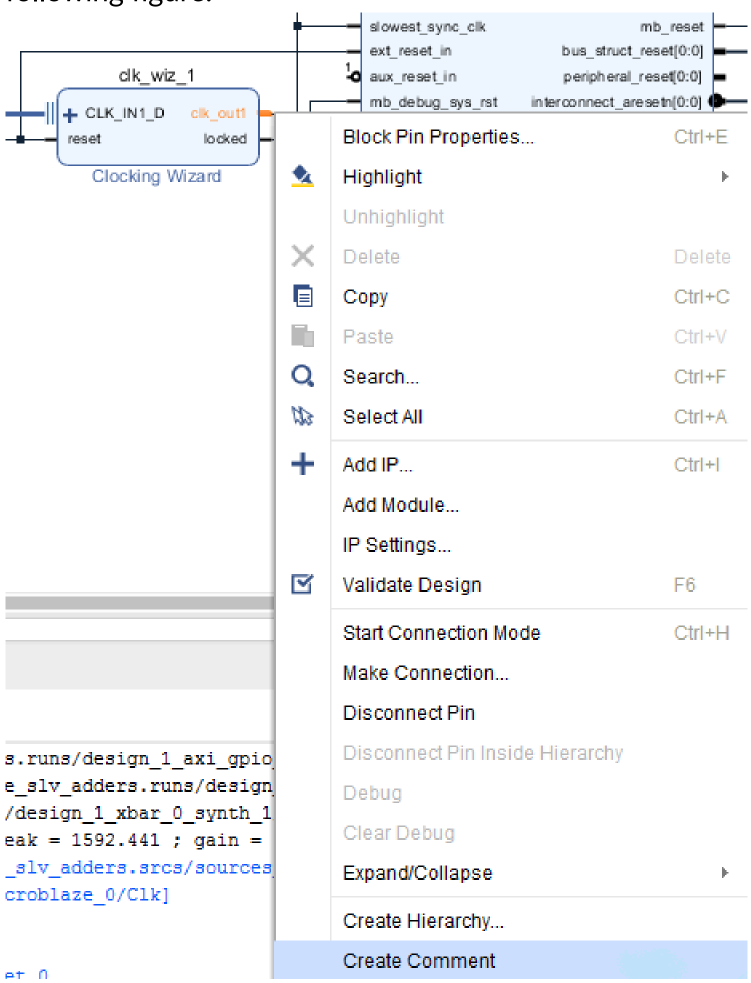
2. Drag and place these comment boxes at any location on the BD canvas.

   These types of “un-anchored” comments are written out at the top of the generated HDL code, as shown in the following figure.



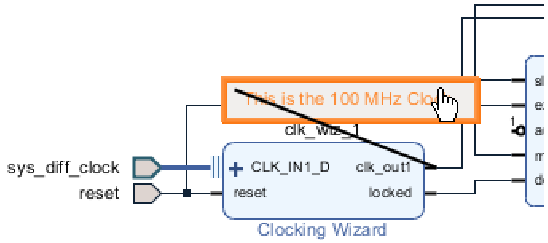
You can also add comments to pins of an IP or to I/O ports in the BD:

1. With the pin or port selected, right-click and select Create Comment, as shown in the following figure.

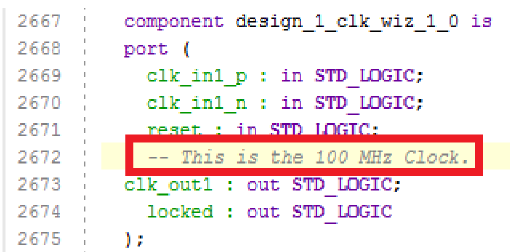


2. In the text box that is created, type your comments.

   This text box can be seen in the GUI as anchored to the pin or port in question, as shown in the following figure.



The generated HDL code contains the comments for that particular pin or port, as shown in the following figure.



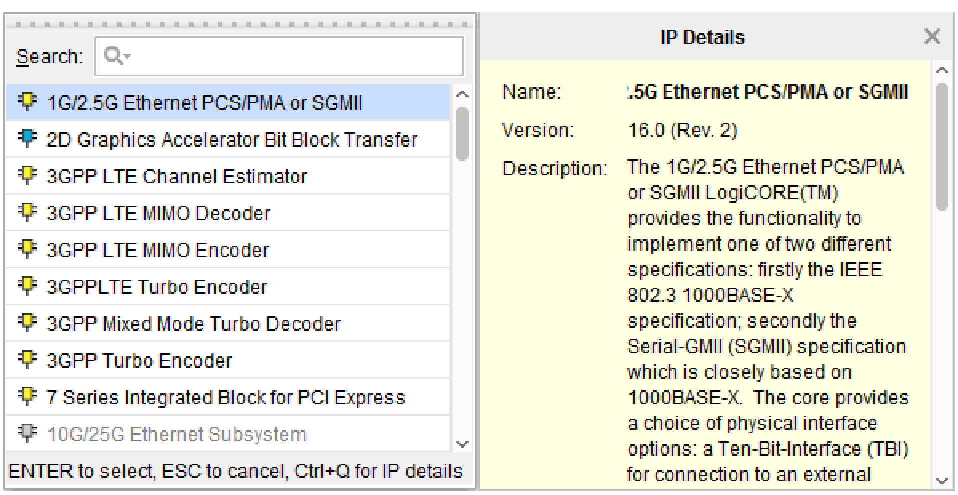
CAUTION! *You can add comments to either pins/ports or interface pins/ports in the GUI. Comments for the pins/ports are written out in the generated HDL. However, comments for interface pins/ports do not appear in generated HDL code.*

Adding IP Modules to the Design Canvas

You can add IP modules to a diagram in the following ways:

1. Right-click in the diagram, and select **Add IP**.

   A searchable IP catalog opens, as shown in the following figure.



TIP: *To enable the IP Details window of the IP catalog, press* **Ctrl+Q** *in the IP catalog window.*

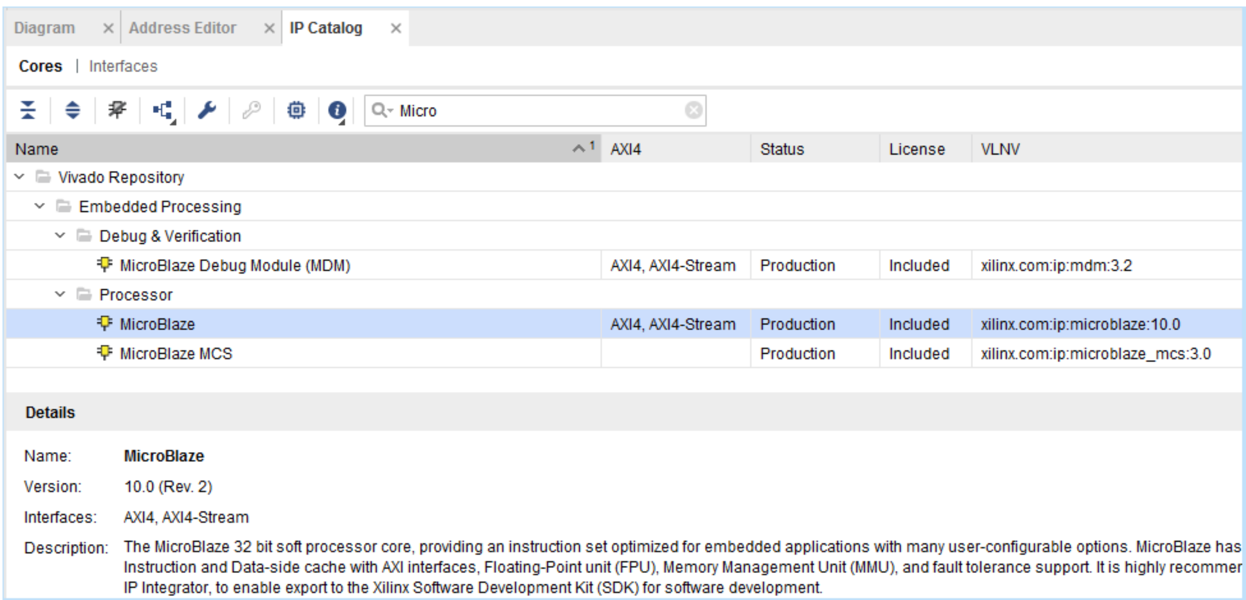
2. Type in the first few letters of the IP name in the Search filter at the top of the catalog.

   IP modules matching the search string display.

3. Select the IP to add by doing one of the following:

   - To add a single IP, you can either click the IP name, and press **Enter** on your keyboard, or double-click the IP name.
   - To add multiple IP to the BD, you can highlight the additional desired IP (**Ctrl**+ click), and press **Enter**.

   You can also add IP to the BD by opening the IP catalog from the Project Manager menu in the Flow Navigator. Use the Search field to find specific IP in the IP catalog window as well.



4. Double-click on a listed IP to add it to the open BD.
5. Float the IP catalog by clicking the **Float** button at the upper-right corner of the catalog window. Then drag and drop the IP of your choice from the IP catalog in the BD canvas.

TIP: *Different fields associated with an IP such as Name, Version, Status, License, and Vendor (VLNV) identification can be enabled by right-clicking in the displayed Header column of the IP catalog and enabling and disabling the appropriate fields.*

Multiple IP can be added to the BD canvas at once by selecting multiple IP in the IP catalog and using one of the methods described above.

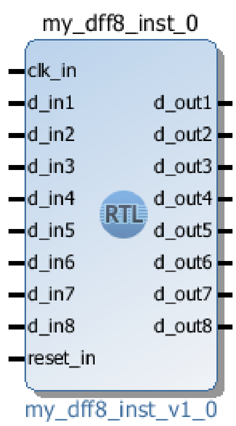
Adding RTL Modules to the Block Design

Using the *Module Reference* feature of the Vivado IP integrator you can quickly add a module or entity defined in an HDL source file directly into your BD. To add an RTL module, the source file must already be loaded into the project, as described at this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

From within the BD select the **Add Module** command from the right-click or context menu of the design canvas. The Add Module dialog box displays a list of all valid modules defined in the RTL source files that you have previously added to the project.

Select one from the list to instantiate it into the BD. The Vivado tools add the module to the BD, and you can make connections to it just as you would with any other IP in the design. The added RTL module displays in the BD with special markings that identify it as an RTL referenced module, as shown in the following figure. This is also referred to as *RTL on Canvas*. See Chapter 13: Referencing RTL Modules, for more information on this feature.

Figure 9: **Modules Referenced from an RTL Source file**



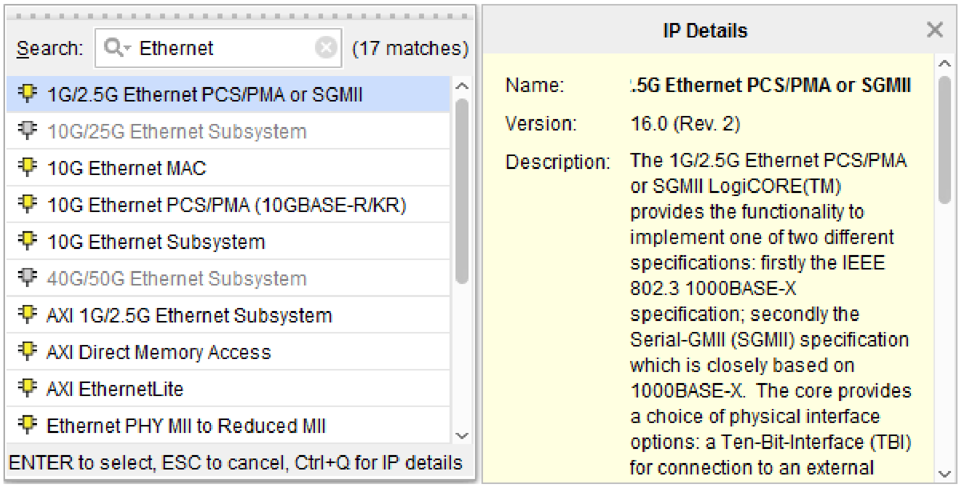
Hierarchical IP in IP Integrator

Some IP in the IP catalog are hierarchical, and offer a child BD inside the top-level BD to display the logical configuration of the IP. These hierarchical IP (also called subsystem IP) let you see the contents of the block, but do not let you directly edit the hierarchy.

Changes to the child BD can only be made by changing the configuration of the IP in the Re-customize IP dialog box.

For example, the 10G Ethernet Subsystem and AXI 1G/2.5G Ethernet Subsystem is an Hierarchical IP in the Vivado IP catalog. You would instantiate these IP just as any other IP by searching and selecting the IP. The following figure shows the 10G Ethernet Subsystem and AXI 1G/2.5G Ethernet Subsystem information.

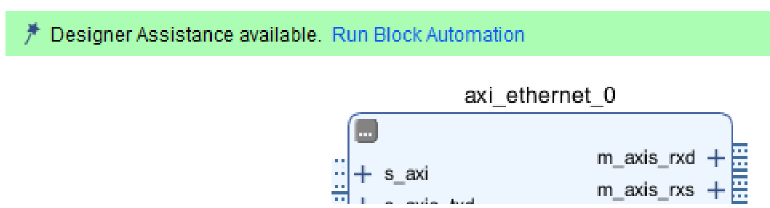
Figure 10: **Adding Hierarchical IP to the Block Design**



When the IP has been instantiated into a BD, double-click the IP to open the Re-customize IP dialog box where you can configure the IP parameters.

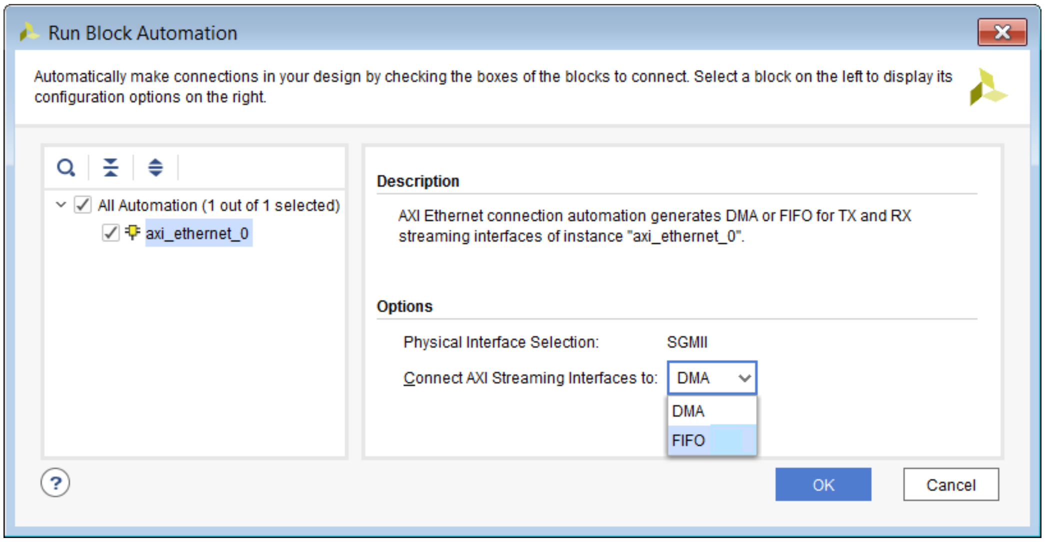
You can run Block Automation for Hierarchical IP when available. This feature creates a subsystem consisting of IP blocks needed to configure the IP.

Figure 11: **Running Block Automation for Hierarchical IP**



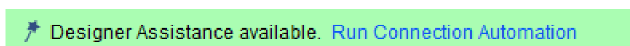
Using the Run Block Automation dialog box, you can select various parameters of the IP subsystem to create. This puts together an IP subsystem for the mode selected, like the one shown in the following figure.

Figure 12: **Run Block Automation Dialog Box**



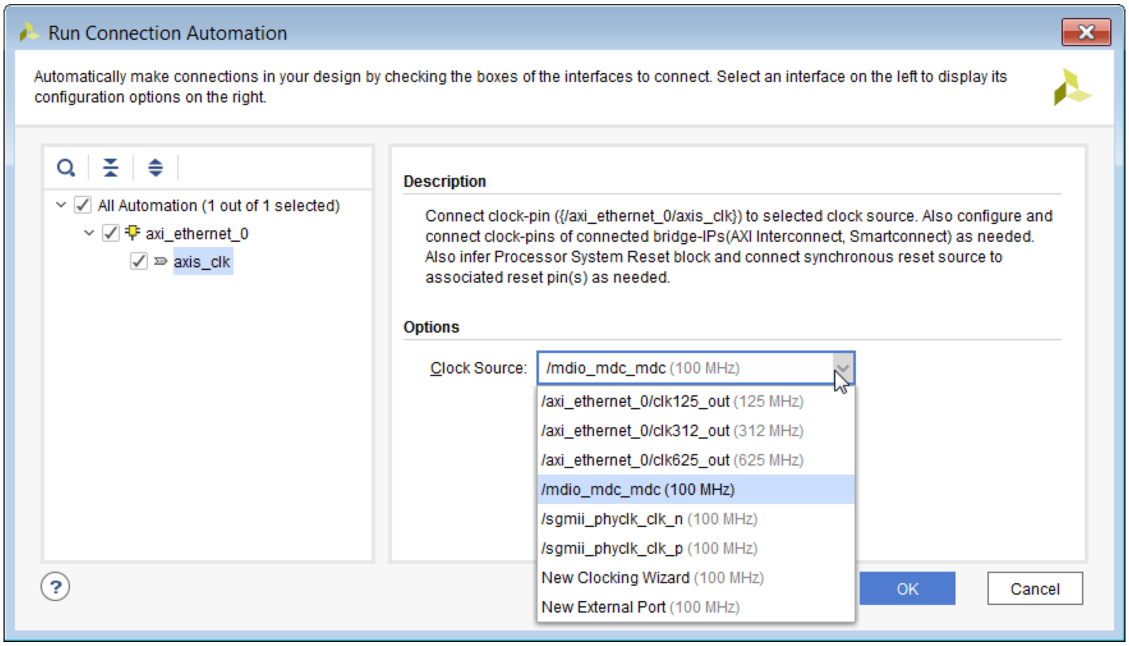
You can also run Connection Automation when it is available to complete connections to I/O ports needed for Hierarchical IP subsystems.

Figure 13: **Running Connection Automation for Hierarchical IP**



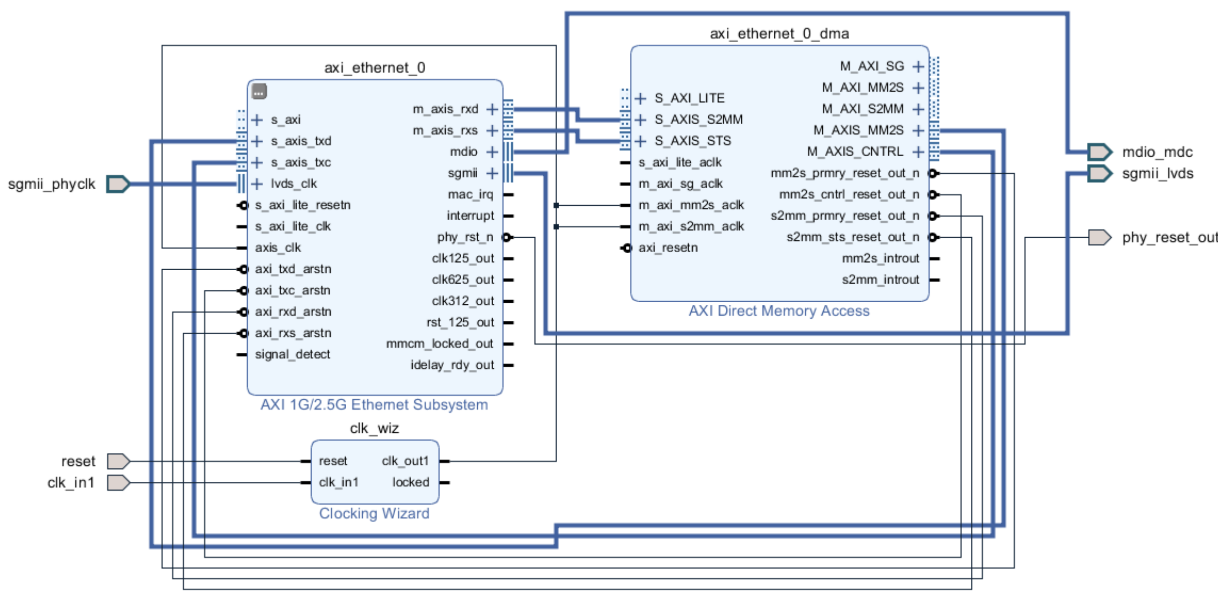
The Run Connection Automation dialog box lets you select different connectivity options for the subsystem. See the following figure.

Figure 14: **Run Connection Automation Dialog Box**



The complete hierarchical IP subsystem should look as shown in the following figure.

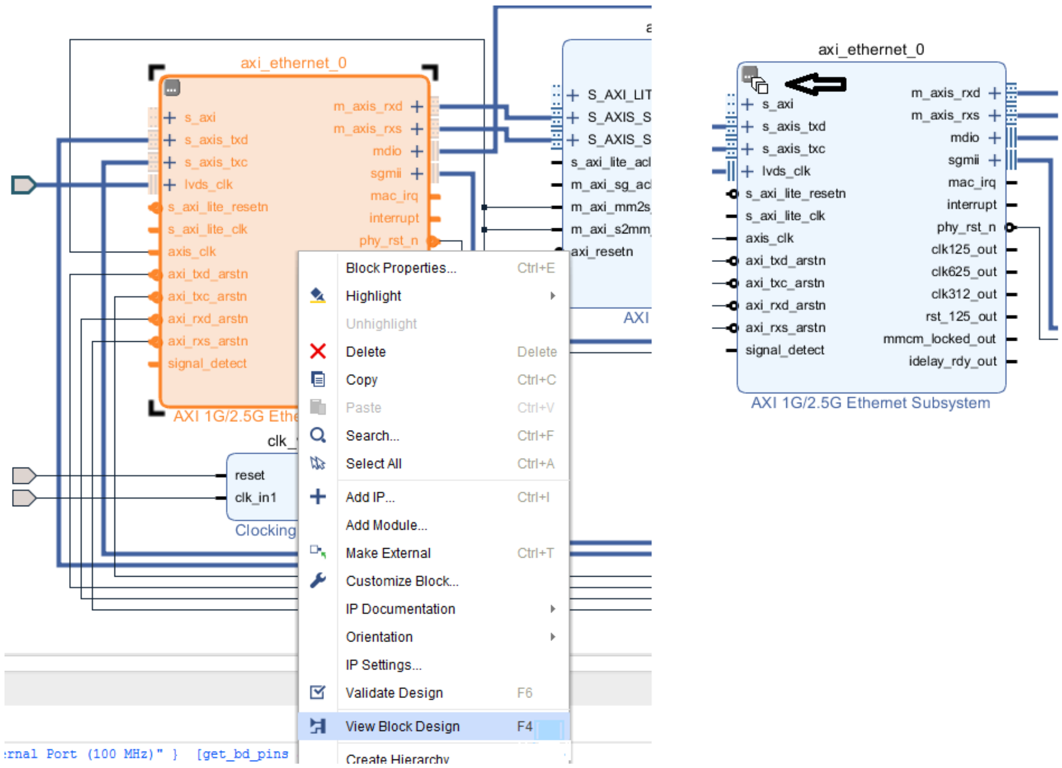
Figure 15: **Hierarchical IP Subsystem After Runnning Designer Assistance**



To view the child BD inside the AXI Ethernet subsystem IP, right-click and select **View Block Design** command, as shown in the following figure.

TIP: *You cannot directly edit the subsystem block design of a Hierarchical IP.*

Figure 16: **View Block Design**

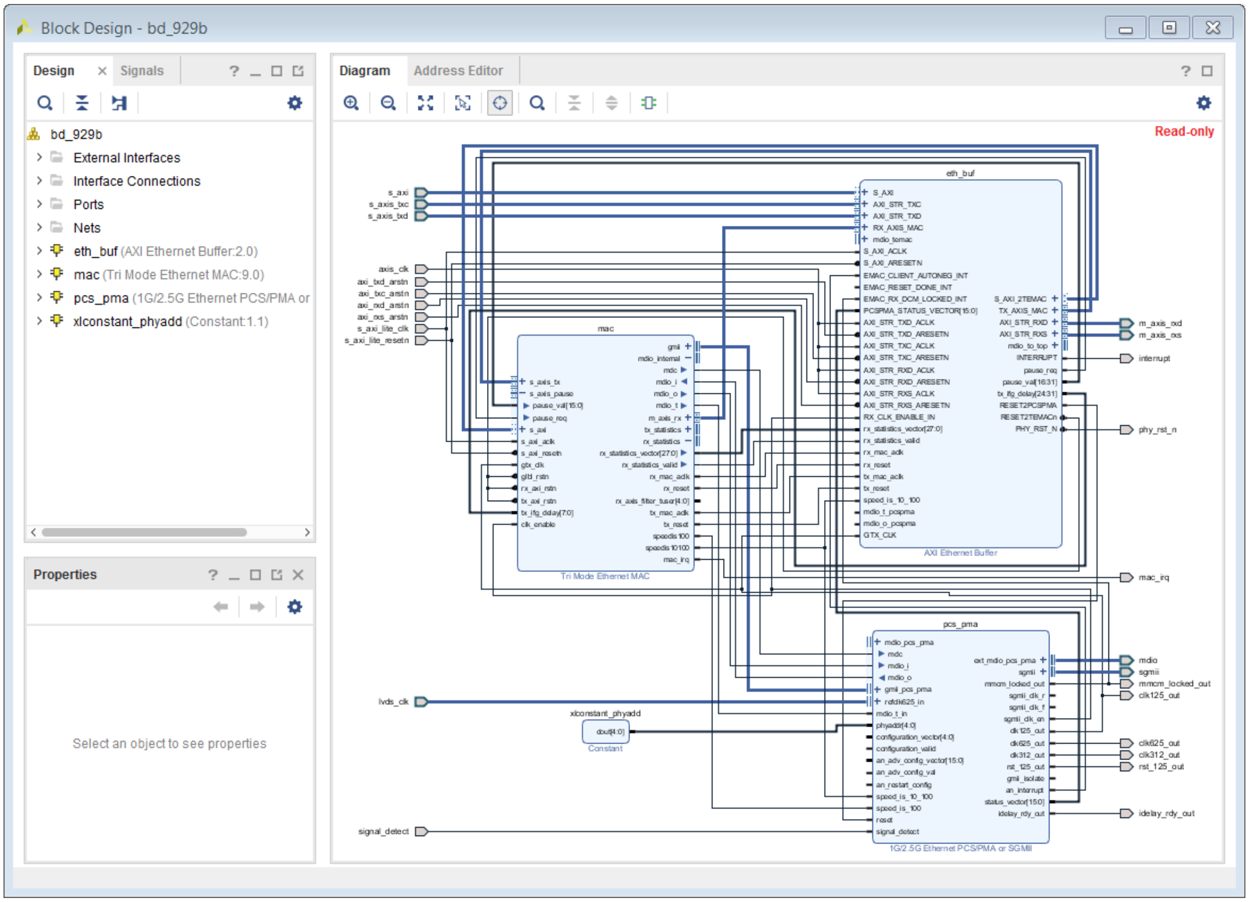


TIP: *If you re-customize the IP while the child-level block design is open, the child-level block design will close.*

To view the BD, click View Block Design icon at the top left corner of the IP symbol.

This opens a BD window showing the child-level BD, as shown in the following figure.

Figure 17: **Child Block Design in Hierarchical IP**



InterConnect vs. SmartConnect

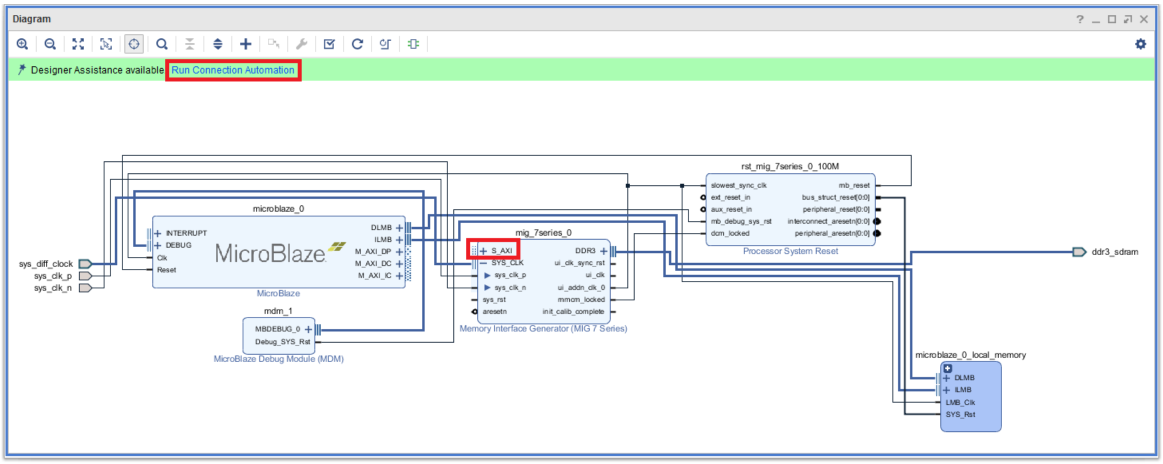
The Xilinx® LogiCORE IP AXI InterConnect and SmartConnect cores both connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices; however, the SmartConnect is more tightly integrated into the Vivado design environment to automatically configure and adapt to connected AXI master and slave IP with minimal user intervention. The AXI Interconnect can be used in all memory-mapped designs.

There are certain cases for high bandwidth application where using a SmartConnect provides better optimization. The SmartConnect IP delivers the maximum system throughput at low latency by synthesizing a low area custom interconnect that is optimized for important interfaces.

The IP Integrator provides the user a choice to select between the AXI InterConnect and a SmartConnect if the endpoints being connected are AXI4 memory-mapped endpoints.

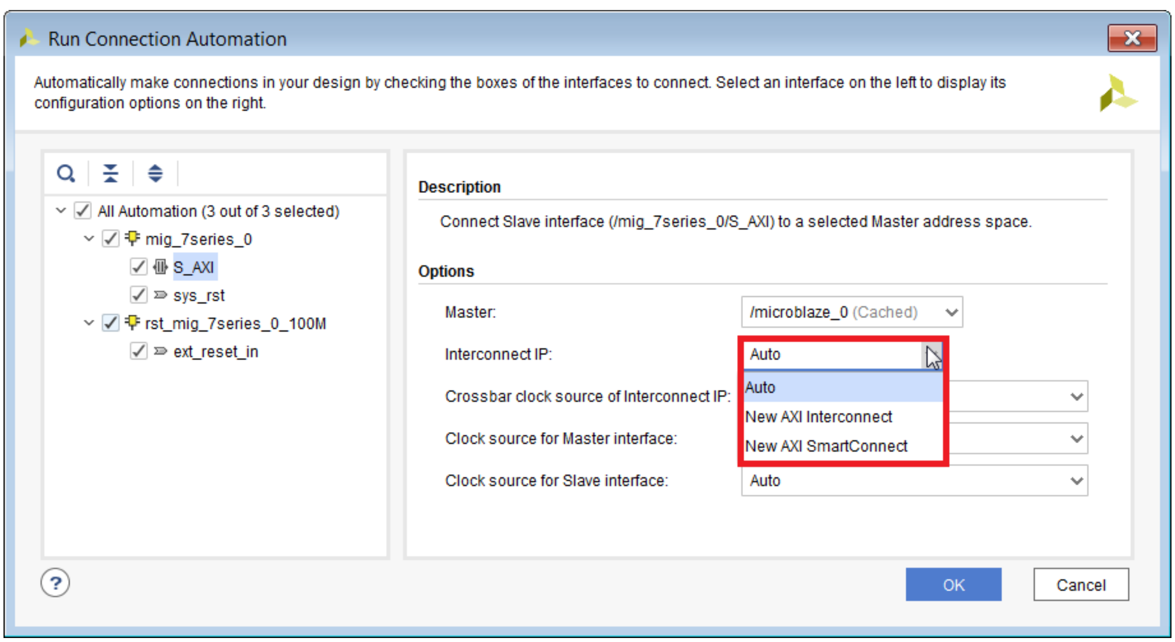
As an example, consider the design example shown in the following figure, where a memory interface IP needs to be connected to a MicroBlaze processor.

Figure 18: **Connecting to High Bandwidth Interfaces**



When you click the Run Connection Automation link, shown in the following figure, the connection automation provides a choice to instantiate either a InterConnect or a SmartConnect, shown in the following figure.

Figure 19: **Run Connection Automation Dialog Box Provides Option to Connect to SmartConnect**



Leaving it to the default selection of Auto instantiates a SmartConnect IP to connect the MicroBlaze processor to the Memory Interface IP.

Glue Logic IP in IP Integrator

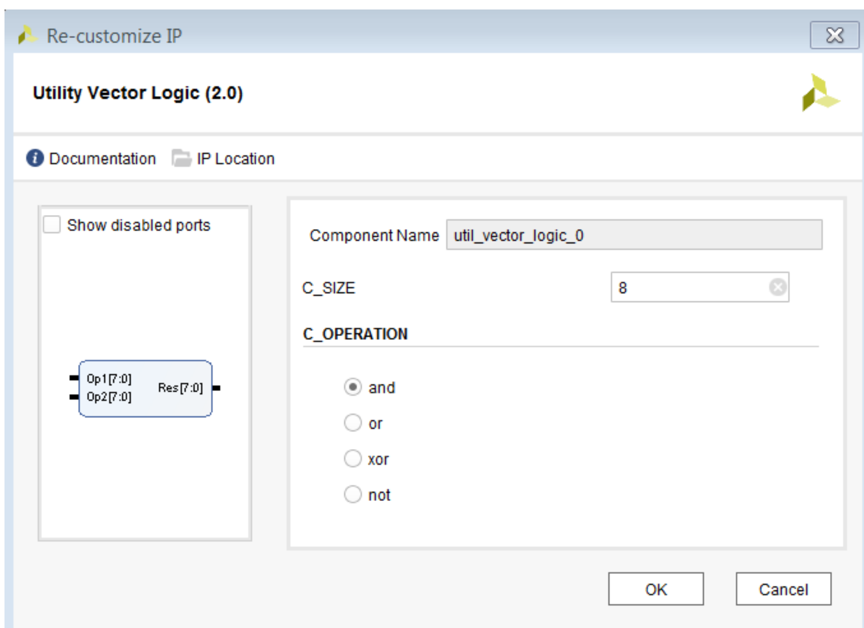
There are several IP available in the IP catalog for use in Vivado IP integrator designs as interconnect or *glue logic*. The following section briefly describes these IP, with references to their product briefs for more information.

Utility Vector Logic

This IP can be configured for different logic modes and input widths. The supported logic operations are `AND`, `OR`, `XOR`, and `NOT`. The `C_Size` is the vector size of the input and output signals, and can be 1 or more. As an example, if the IP is configured in the `AND` mode and `C Size` is set to 4, then the resulting logic would consist of 4 parallel, 2-input `AND` gates.

If the IP is configured as an inverter or `NOT`, then the `C_Size` denotes the number of single bit inverters. See the *LogiCORE IP Utility Vector Logic Product Brief* (PB046) for more information.

Figure 20: **Utility Vector Logic IP Dialog Box**

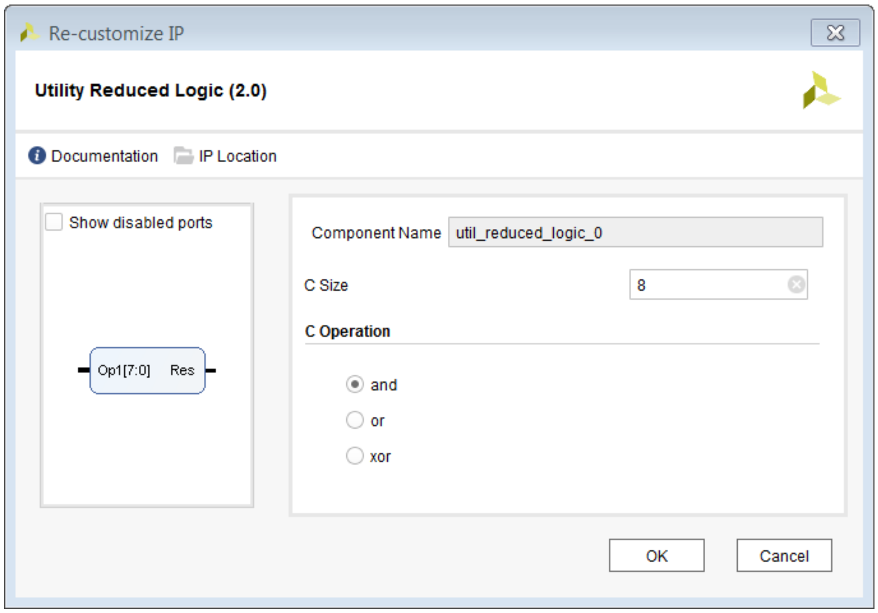


Utility Reduced Logic

This IP can be configured as `AND`, `OR`, and `XOR` functions. `C_Size` sets the number of inputs to the function, and must be at least 2. Refer to the *LogiCORE IP Utility Reduced Logic Product Brief* (PB045) for more information.

For example, setting the `C_Size` to 8 as an `AND` function creates one 8 input `AND` gate, with a single output, shown in the following figure.

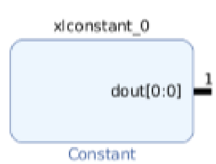
Figure 21: **Utility Reduced Logic IP Dialog Box**



Constant

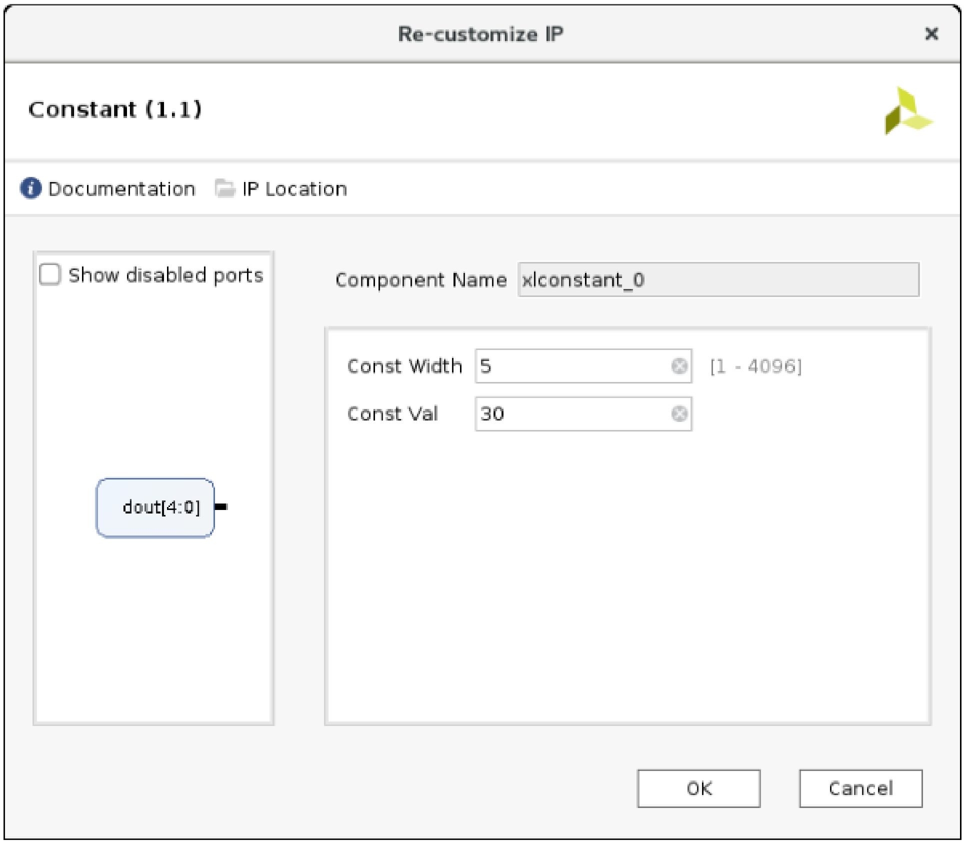
Use the Constant IP to tie signals up or down, and specify a constant value. The Constant IP shows the constant value being driven by the block on the output pin. As an example, the following constant IP shows a default value on the output pin `dout[0:0]` being driven to **1**.

Figure 22: **Constant IP with default value**



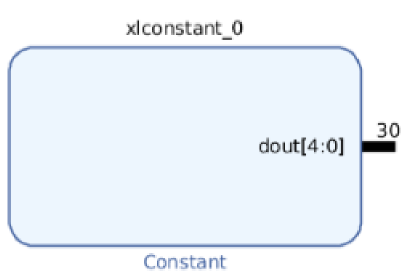
To change the value double click the IP. This brings the configuration dialog box as shown.

Figure 23: **Constant IP Configuration dialog box**



In this configuration dialog box, the width of the value to be driven is set to **5** (binary bits) and a value of decimal **30** is being driven. After the user commits to the changes, the Constant IP shows the new values being driven on the output pin `dout[4 downto 0]`.

Figure 24: **Constant IP after changing the constant value being driven**



Constant values can be set as decimal, hexadecimal or binary values. See the *LogiCORE IP Constant Product Brief* (PB040) for more information.

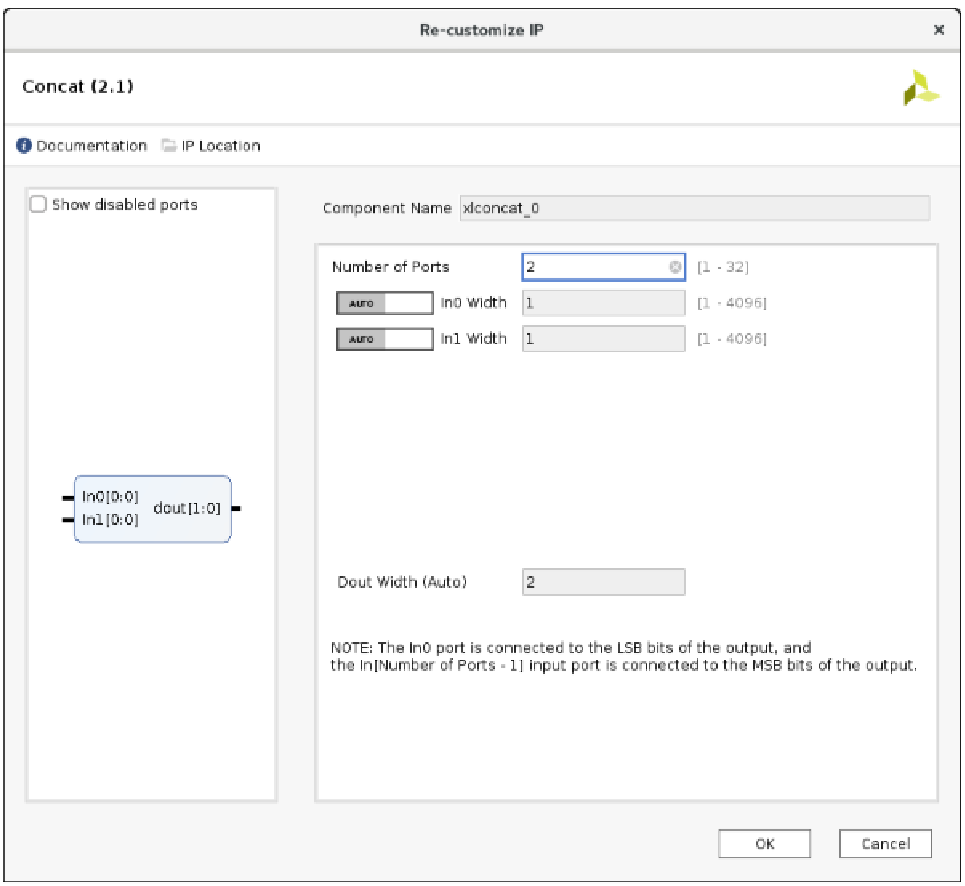
Utility Buffer

There are occasions when you need to manually insert a clock or signal buffer into a BD. You can use the Utility Buffer IP in these situations to configure and instantiate one of several different buffer types into the design. See the *LogiCORE IP Utility Buffer Product Brief* (PB043) for more information.

Concat

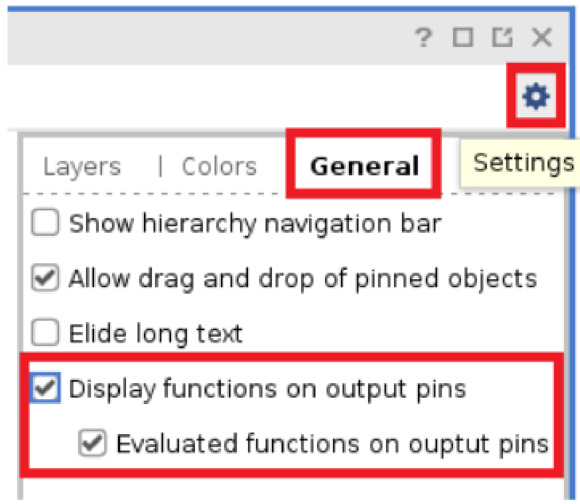
To combine or concatenate bus signals of varying widths, use the Concat IP. The Number of Ports defines the number of source signals that need to be concatenated together. Each of source can be of different width, as automatically determined by IP integrator or user-specified, as shown in the following figure. The resulting output is a bus that combines the source signals together. See the *LogiCORE IP Concat Product Brief* (PB041) for more information.

Figure 25: **Concat IP Dialog Box**



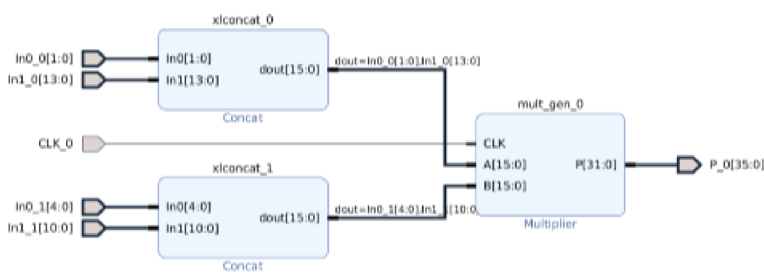
The output pin of the Concat IP, can be enabled to display the concat values. To enable the display of the pins being concatenated, click the **Settings** button on the top right corner of the block design canvas. Select the General Tab and check the boxes against **Display functions on output pins** and **Evaluate functions on output pins**.

Figure 26: **Enabling display of the inputs being concatenated**



As a simple illustration Concat IP blocks are being used to drive a Multiplier IP.

Figure 27: **Concat IP Example**



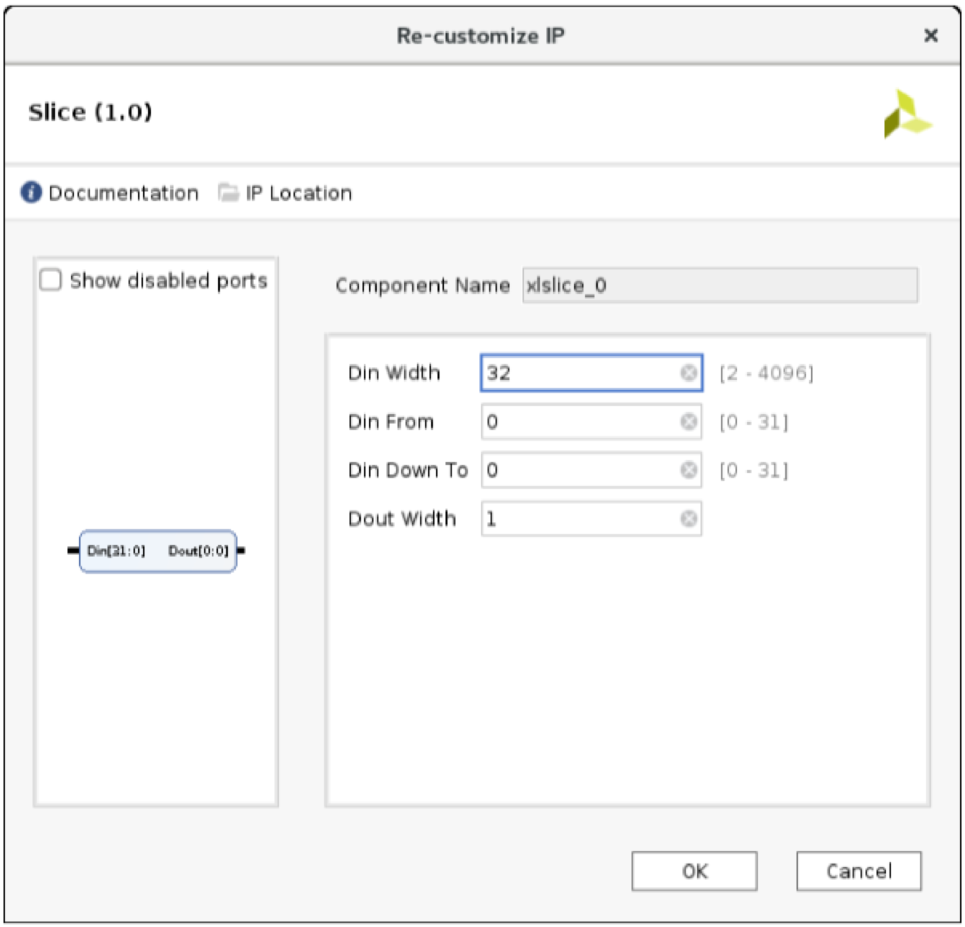
As can be seen, the multiplier, `mult_gen_0`, has two inputs `A` and `B` which are both 16-bits wide. The Concat IP `xlconcat_0` and `xlconcat_1` instances drive the 16 bits out on the output pin `dout[15:0]`. The `dout[15:0]` pin on the `xlconcat_0` instance concatenates two inputs `In0_0[1:0]` and `In1_0[13:0]`. This concatenated value can be seen on the output pin of the `xlconcat_0` block `dout[15:0]`. Similarly, the `xlconcat_1` instance concatenated value can be seen on the output pin `dout[15:0]`.

***Note*:** Note that the evaluated functions cannot be displayed on the output pin until connectivity of the output pin is made to a destination pin on the design.

Slice

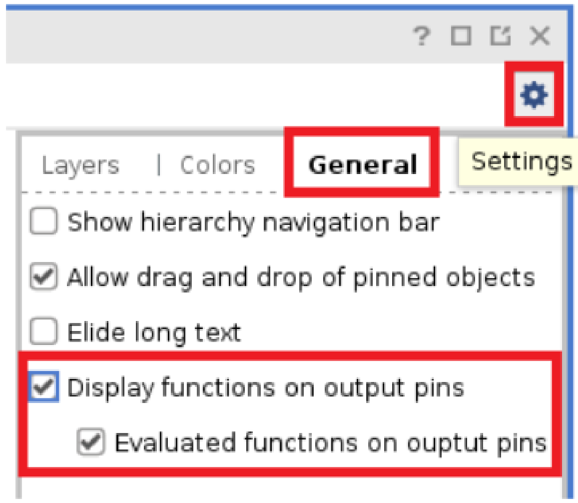
To rip bits out of a bus signal, use the Slice IP. The Din Width field specifies the width of the input bus, and Din From and Din Down To fields specify the range of bits to rip out. The output width, Dout Width, is automatically determined. See the *LogiCORE IP Slice Product Brief* (PB042) for more information.

Figure 28: **Slice IP Dialog Box**



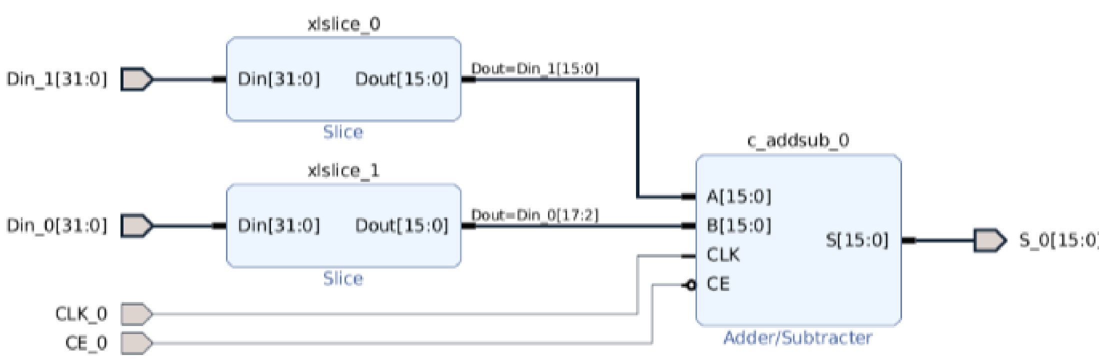
The output pin of the Slice IP, can be enabled to display the bits being ripped off from a bus. To enable the display of the pins being ripped, click the **Settings** button on the top right corner of the block design canvas. Select the General Tab and check the boxes against **Display functions on output pins** and **Evaluate functions on output pins**.

Figure 29: **Enabling display of the bits being ripped off**



As a simple illustration Slice IP blocks are being used to drive the input pins of a Adder/Subtractor IP.

Figure 30: **Slice IP Example**



As can be seen, the Adder/Subtractor IP, `c_addsub_0`, has two inputs `A` and `B` which are both 16-bits wide. The Slice IP `xslice_0` and `xslice_1` instances drive the 16 bits out on the output pin `Dout[15:0]`. The `dout[15:0]` pin on the `xslice_0` instance rips 16 bits [bits 15 through bit 0], off of the 32-bit input buts `Din0_0[31:0]`. This "ripped-off" value can be seen on the output pin of the `xlslice_0` block `dout[15:0]` as Dout=Din_1[15:0]. Similarly, the ripped output of `xlslice_1` instance, Dout=Din_0[17:2], can be seen on the output pin `dout[15:0]`.

***Note*:** Note that the evaluated functions cannot be displayed on the output pin until connectivity of the output pin is made to a destination pin on the design.

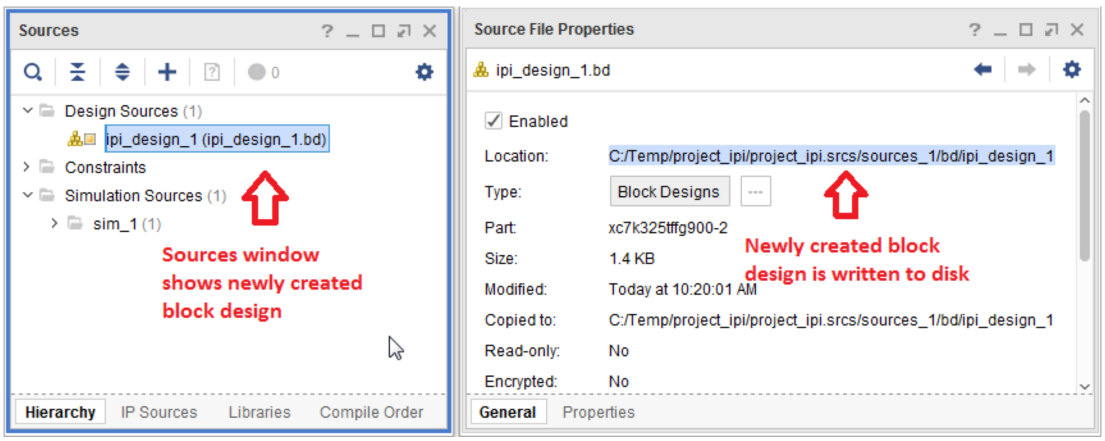
TIP: *You can use multiple Slice IP to pull different widths of bits from the same bus net.*

About On-Disk Objects and In-Memory Objects

Block Designs

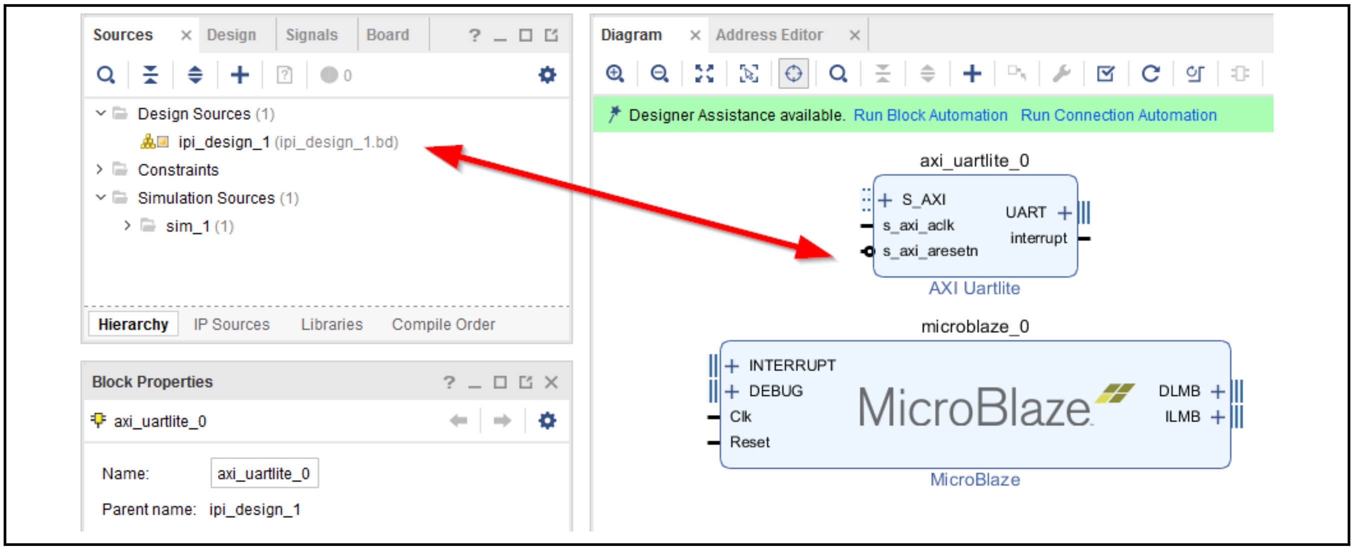
Block Designs (BDs) are *on-disk* objects. When you create a BD, it gets written to the disk. Accordingly, the Sources window shows the creation of the BD, shown in the following illustrated figure.

Figure 31: **Sources Window and Properties View of Block Design upon Creation**



IP Instances or Block Design Cells

IP instances or cells on the BD are *in-memory* objects. That is a copy of the instantiated IP is created in memory, but it is *not* written to disk until you save the BD.

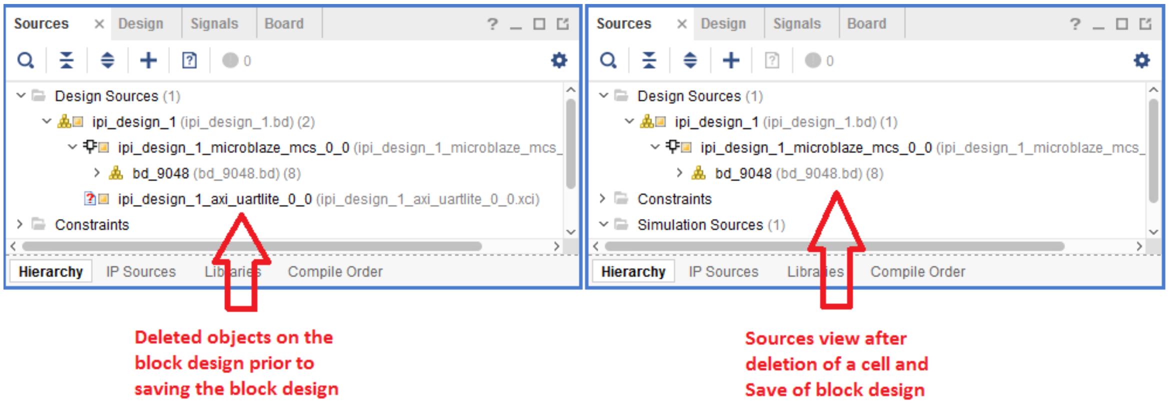


As can be seen in the following figure, as cells (IP) are instantiated in the block design, they do not appear in the Sources window under the BD. At this point all cells or IP objects are created in-memory. The same applies to Hierarchical IP or IP Subsystems. The IP and the related files, such as BDs underneath IP subsystems, sub-cores, and so forth, are not written to disk until you save the BD. After you save the BD, the Sources window is updated to show all the IP under the BD hierarchy as shown in the following figure.

Figure 32: **Sources Window View of IP or Cells After Saving the Block Design**

After saving the BD, if you delete an IP from the BD canvas, the Sources window shows the IP sources with a "?" icon. This updates after you save the BD. (See the following figure.)

Figure 33: **Sources Window View After Deleting a Cell Before/After a Save**



Validating a Block Design

You can validate a BD either before saving or after saving it. Use the Validate Button for easy access.

Generating/Resetting Output Products

You can generate or reset output products with or without saving the BD; however, these operations perform an automatic save.

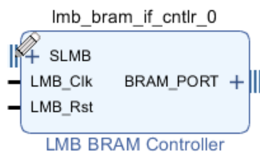
Making Connections

When you create a design in IP integrator, you add blocks to the diagram, configure the blocks as needed, make interface-level or simple-net connections, and add interface or simple ports.

Making connections in IP integrator is designed to be simple. As you move the cursor near an interface or pin connector on an IP block, the cursor changes into a pencil. You can then click an interface or pin connector in an IP block, hold down the left-mouse button, and then draw the connection to the destination block.

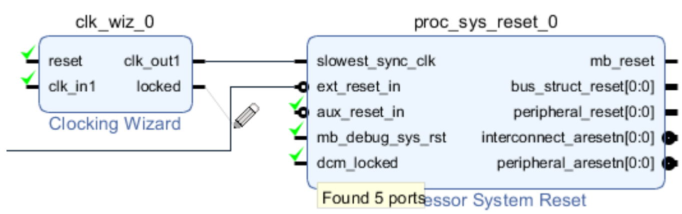
A signal or bus-level connection is shown as a narrow connection line on a symbol. Buses are treated identically to individual signals for connection purposes. An interface-level connection is indicated by a more prominent connection box on a symbol, as shown on the SLMB interface pin in the following figure.

Figure 34: **Pins on a Symbol**



When you are making connections, a green check mark appears next to any compatible destination connections, highlighting the potential connections for the signal or interface.

Figure 35: **Signal or Bus Connection on a Symbol**



When signals are grouped as an interface, you can quickly connect all of the signals and buses of the interface with other compatible interface pins. The compatible interfaces are also identified by a green check mark.

Connecting Interface Signals

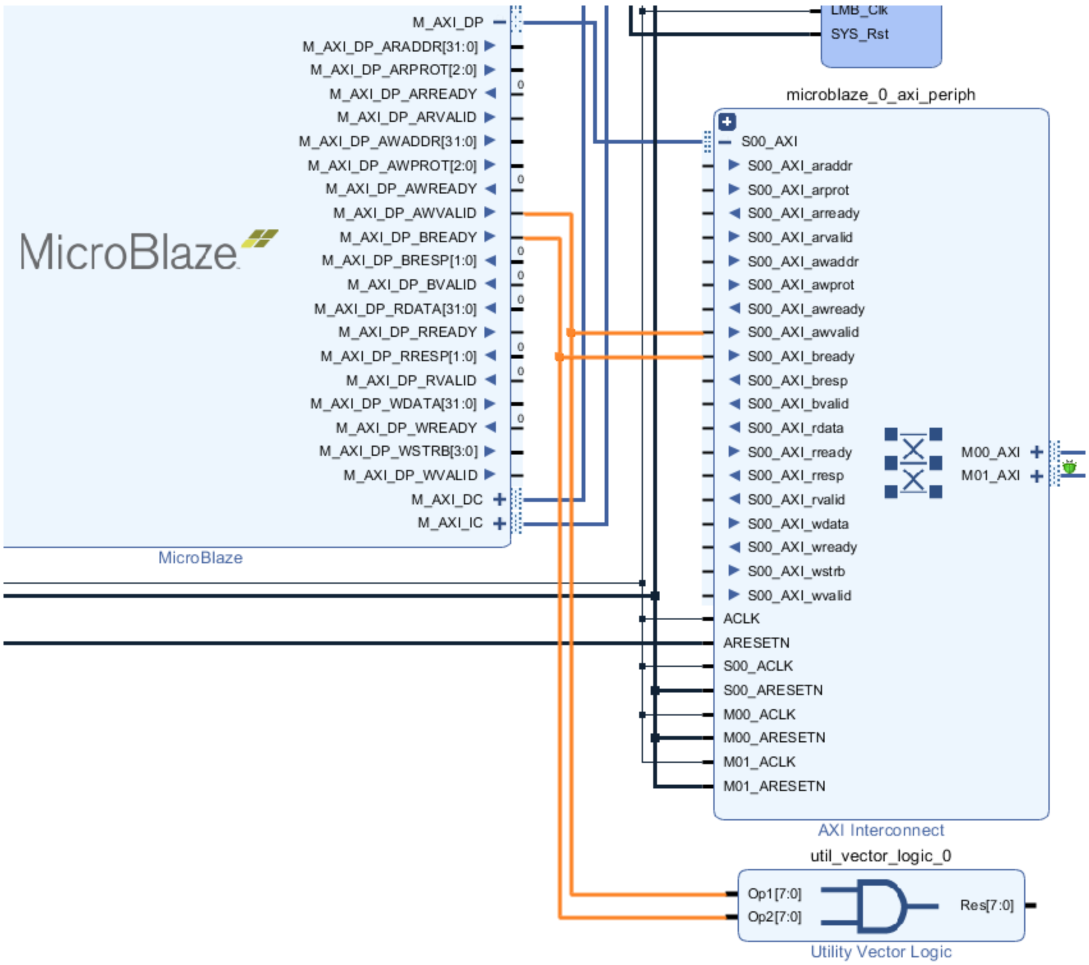
To connect to the individual signals or buses that are part of an interface pin, you can expand the interface pin to display those individual signals. Clicking the + symbol on the interface expands the interface to display its contents.

In the following figure, you can see that the interface pin `M_AXI_DP` on the `microblaze_0` instance is connected to the `S00_AXI` interface pin on the `microblaze_0_axi_periph` instance. In addition, two individual signals of the interface (`AWVALID` and `BREADY`) are connected to a third instance, `util_vector_logic_0`, to `AND` the signals.

When individual signals of an interface are separately connected from the rest of the interface, the signals must include all of the pins needed to complete the connection. In the example shown in the following figure, both the master and slave AXI interface pins are expanded to enable connection to the individual `AWVALID` and `BREADY` signals, as well as connecting to the pins of the Utility Vector Logic cell.

IMPORTANT! *Individually connected interface signals are no longer connected as part of the interface in the BD. The individual signal is essentially removed from the interface. The entire signal must be manually connected.*

Figure 36: **Expanding the Interface to Make a Connection**



When connections to an interface pin are overridden by connection to individual signals or bus pins of the interface, the Vivado tool issues a warning similar to the following:

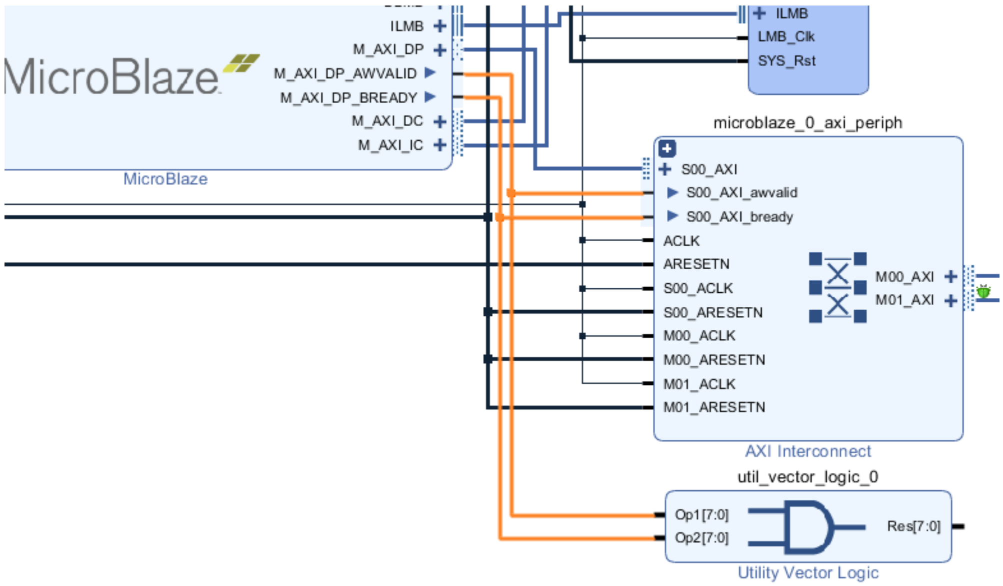
```
WARNING: [BD 41-1306] The connection to interface pin /microblaze_0/
M_AXI_DP_AWVALID
is being overridden by the user. This pin will not be connected as a part
of interface
connection M_AXI_DP
```

This warning should be expected because the connection is no longer be included as a part of the interface, and you must manually complete the connection.

After making connections to signals or buses inside of an interface pin, you can collapse the interface to shrink the block and hide the details of the pin. Clicking on the - symbol on an expanded interface pin collapses it to hide its contents.

As seen in the following figure, the separately connected signals or buses of the interface continue to be shown as needed to properly display the connections of the BD.

Figure 37: **Collapsing the Interface**



External Connections

You can connect signals and interfaces to external I/O ports as follows:

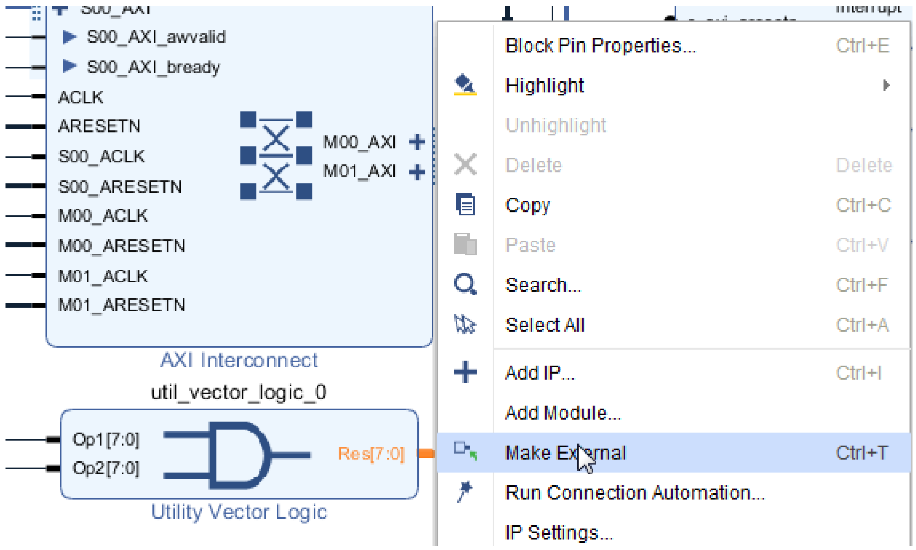
- Making Ports External
- Creating Ports
- Creating Interface Ports

The following sections describe these options.

Making Ports External

1. To connect signals or interfaces to external ports on a diagram, first select a pin, bus, or interface connection, as shown in the following figure.
2. Right-click and select **Make External**.

   You can also select **Ctrl + Click** to select multiple pins and invoke the Make External command for all pins at one time.

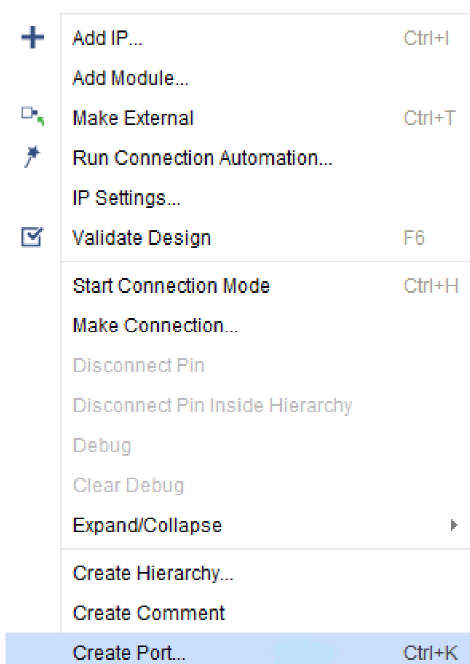


This command ties a pin on an IP to an I/O port on the BD. IP Integrator connects the port on the IP to an external I/O.

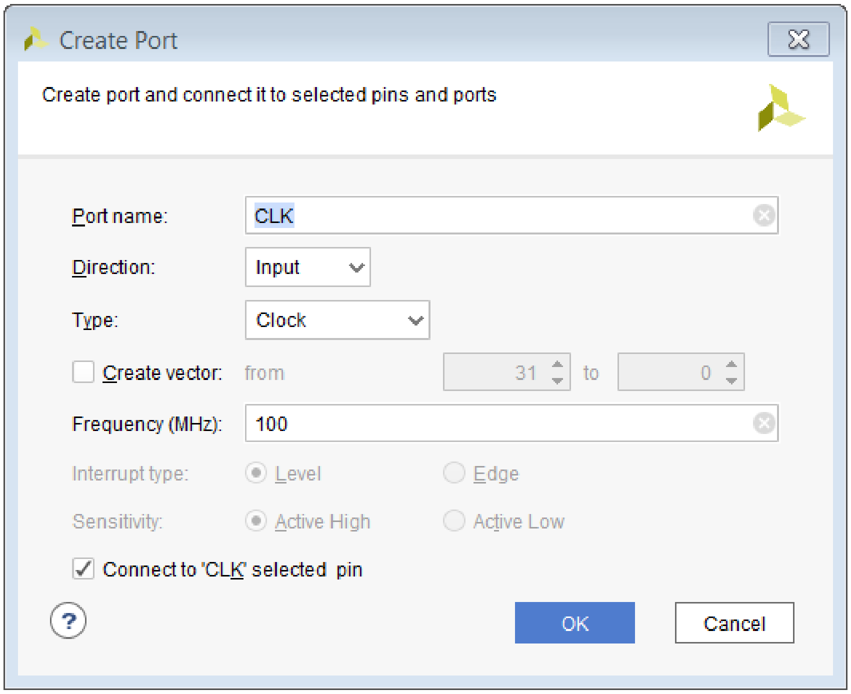
Creating Ports

1. To use the Create Port option, right-click and select **Create Port**, as shown in the following figure.

   This feature is used for connecting individual signals, such as a `clock`, `reset`, and `uart_txd`. The Create Port option gives you more control in specifying the input and output, the bit-width and the type (such as `clk`, `reset`, `interrupt`, `data`, and `clock enable`).



The Create Port dialog box opens, as shown in the following figure.



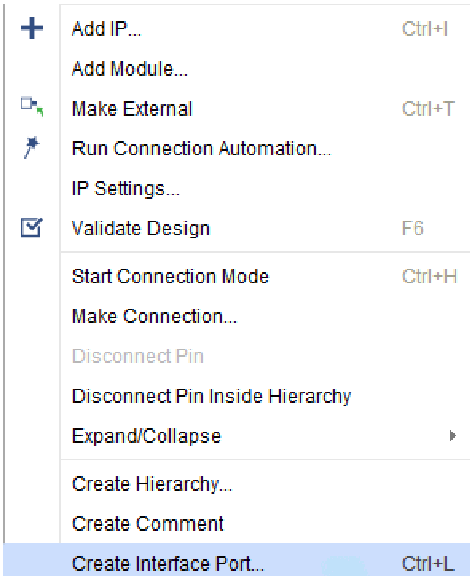
2. Specify the Port name, the Direction, such as input, output or bidirectional, and the Type (such as clock, reset, interrupt, data, clock enable or custom type).

   You can also create a bit-vector by checking the Create Vector field and then selecting the appropriate bit-width. You can also specify the input Frequency, the Interrupt type, and the Polarity, as well as use the Connect to '<pin_name>' selected pin check box.

Creating Interface Ports

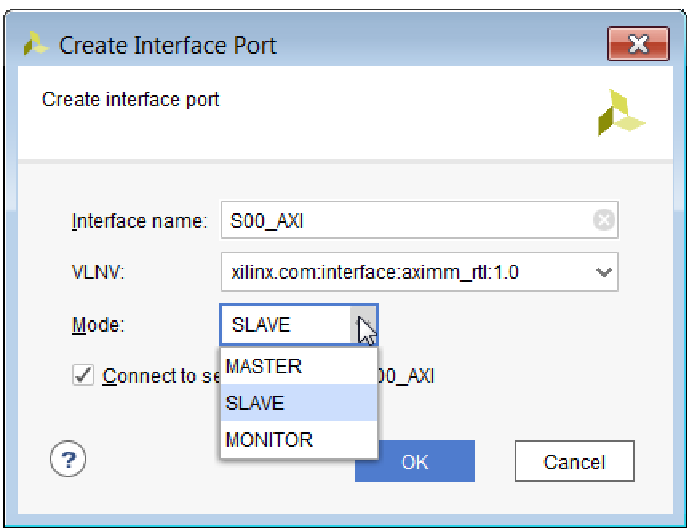
To use the create interface port option:

1. Right-click and select **Create Interface Port**, as shown in the following figure.



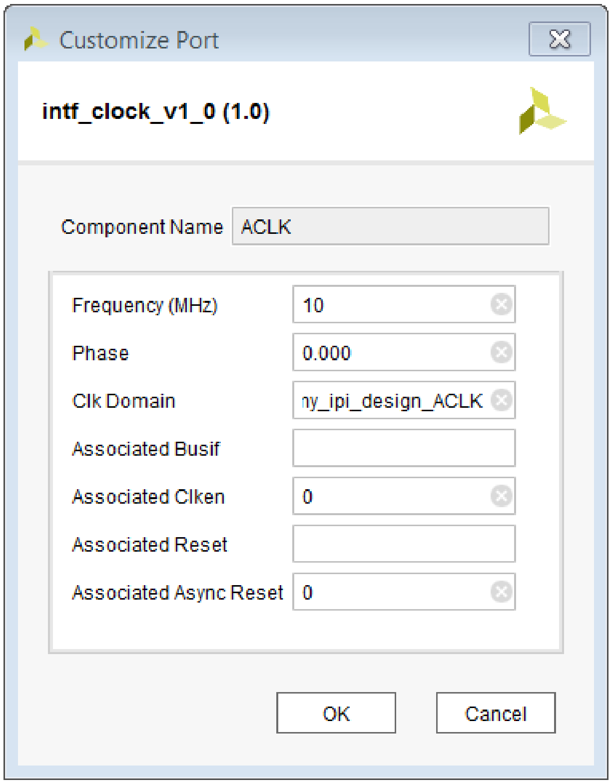
This command creates ports on the interface pins which are groupings of signals that share a common function. The Create Interface Port command gives more control in terms of specifying the interface type and the mode (master/slave).

2. In the Create Interface Port dialog box, shown in the following figure, specify the interface name, the Vendor, Library, Name, and Version (VLNV) field, and the mode field such as MASTER or SLAVE.

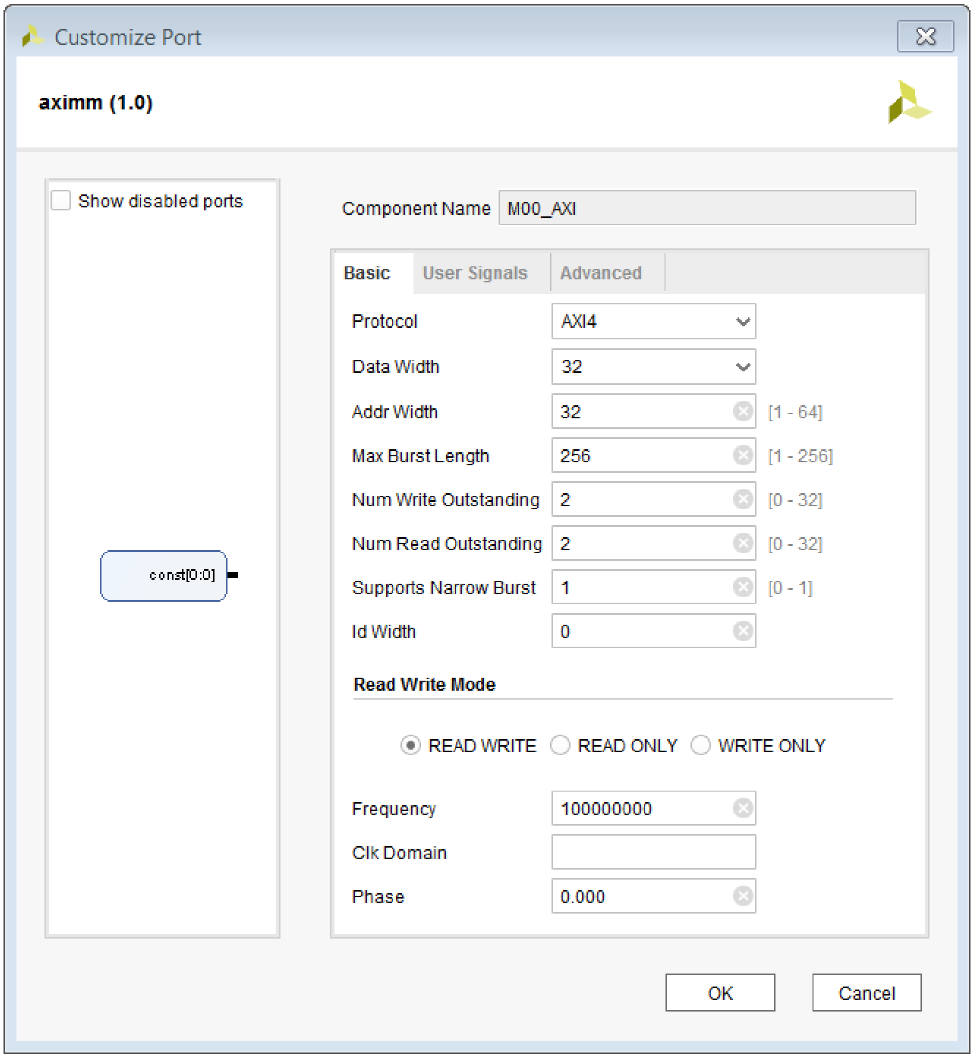


3. Double-click external ports to see their properties, and modify them.

   In the following figure, the port shown is a clock input source, so you can specify different properties, such as frequency, phase, clock domain, any bus interface, the associated clock enable, associated reset and associated asynchronous reset (frequency).



4. On an AXI interface, double-click the port to open the port configuration dialog box.



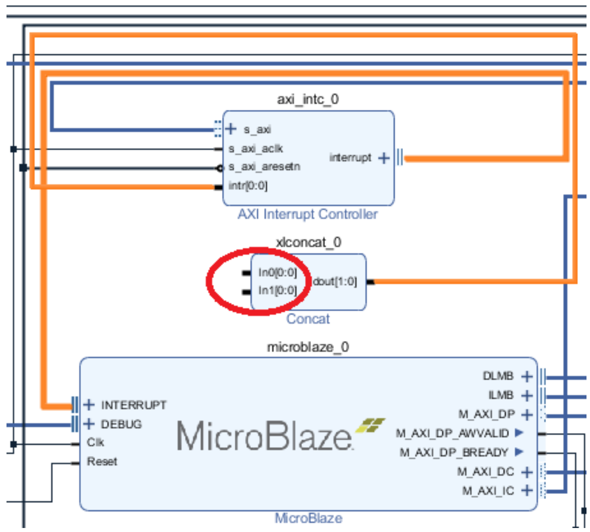
Handling Interrupts

Interrupt handling depends upon the selected processor.

- For a MicroBlaze™ processor, the AXI Interrupt Controller IP must be used to manage interrupts.
- For a Zynq®-7000 SoC processor or the Zynq MPSoC, the Generic Interrupt Controller block within the Zynq processor handles the interrupt.

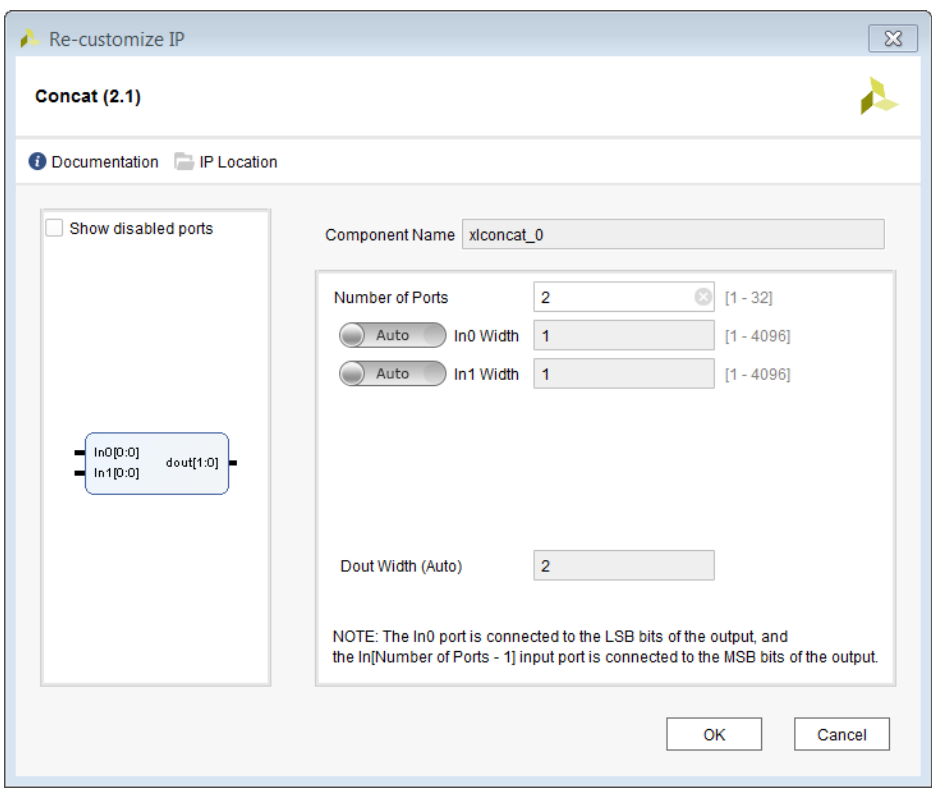
Regardless of the processor used in the design, a Concat IP consolidates and drives the interrupt pins. See the previous Concat section for the brief description provided in this guide.

Figure 38: **Concat IP Driving Interrupt Input to AXI Interrupt Controller**



The inputs of the Concat IP are driven by different interrupt sources. Accordingly, you must configure the Concat IP to support the appropriate number of input ports. Set the Number of Ports field to the number of interrupt sources in the design, as shown in the following figure.

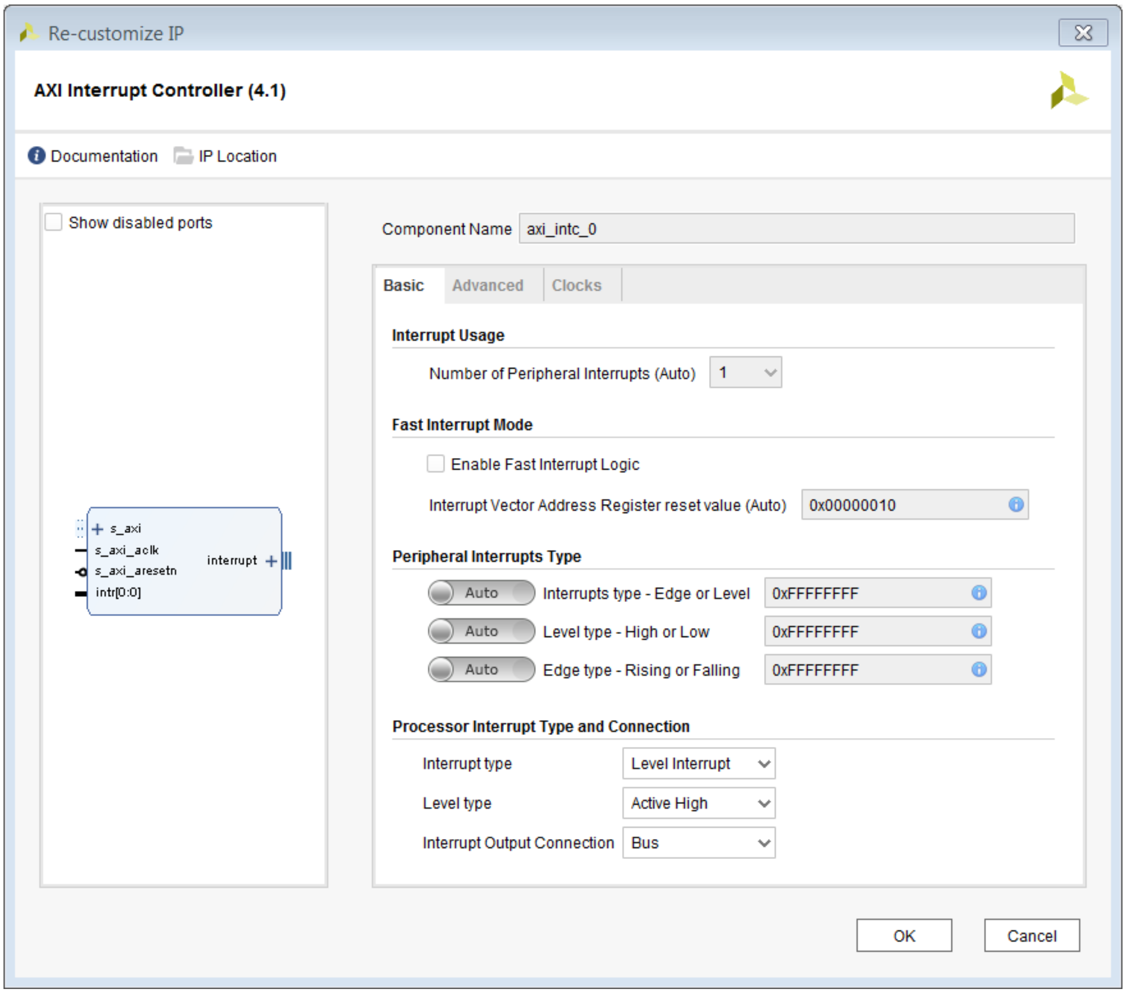
Figure 39: **Concat Re-Customize IP Dialog Box**



TIP: *The width of the output (`dout`) is set automatically during parameter propagation.*

You can configure several of the parameters for the AXI Interrupt Controller. The following figure shows the parameters available from the Basic tab of the AXI Interrupt Controller, of which several are configurable.

Figure 40: **AXI Interrupt Controller Basic Tab Parameters**



- The Number of Peripheral Interrupts is set automatically during parameter propagation and cannot be set by a user. The value is determined by the number of interrupt sources that are driving the inputs of the Concat IP.
- The Fast Interrupt Mode can be set by the user if low latency interrupt is desired.
- The Peripheral Interrupts Type is set to Auto, which can be overridden by the user by toggling the Auto setting to Manual. In manual mode, you can specify the custom values in these fields.
- The Processor Interrupt Type field offers two choices:
    - Interrupt Type

- Level Type or Edge Type, depending on the Interrupt Type setting.

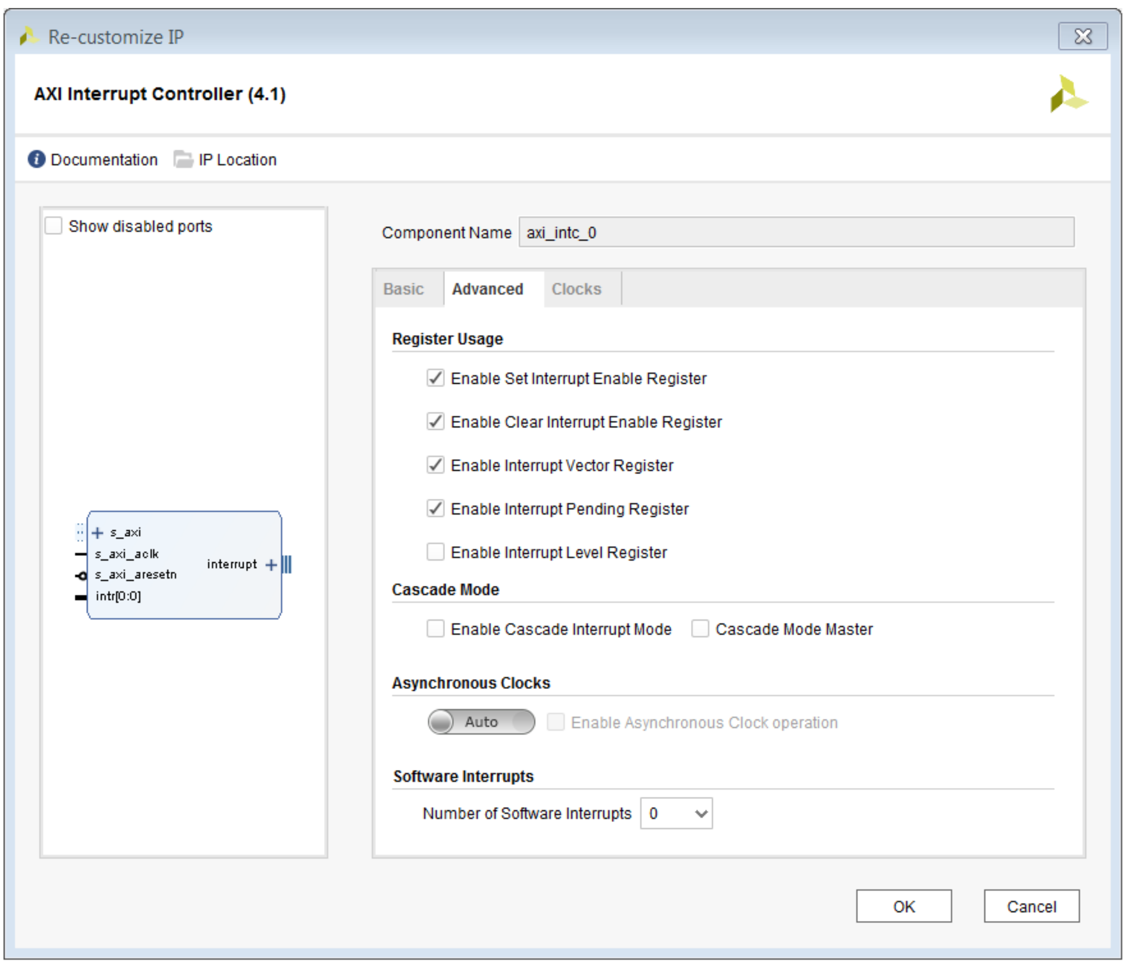
If the Interrupt Type is Edge Interrupt, the other choice is Edge Type. If the Interrupt Type is Level Interrupt, the other choice is Level Type.

You can select if the interrupt source is either Edge-triggered or Level-triggered. Accordingly, you can then also select whether the interrupt is rising or falling edge and, in case of a Level triggered interrupt, the interrupt is active-High or active-Low.

In IP integrator, this value is normally automatically determined from the connected interrupt signals, but can be set manually.

The following figure shows parameters on the Advanced tab of the AXI Interrupt Controller. See the *AXI Interrupt Controller (INTC) LogiCORE IP Product Guide* (PG099) for details of these parameters.

Figure 41: **Interrupt Controller Advanced Tab**



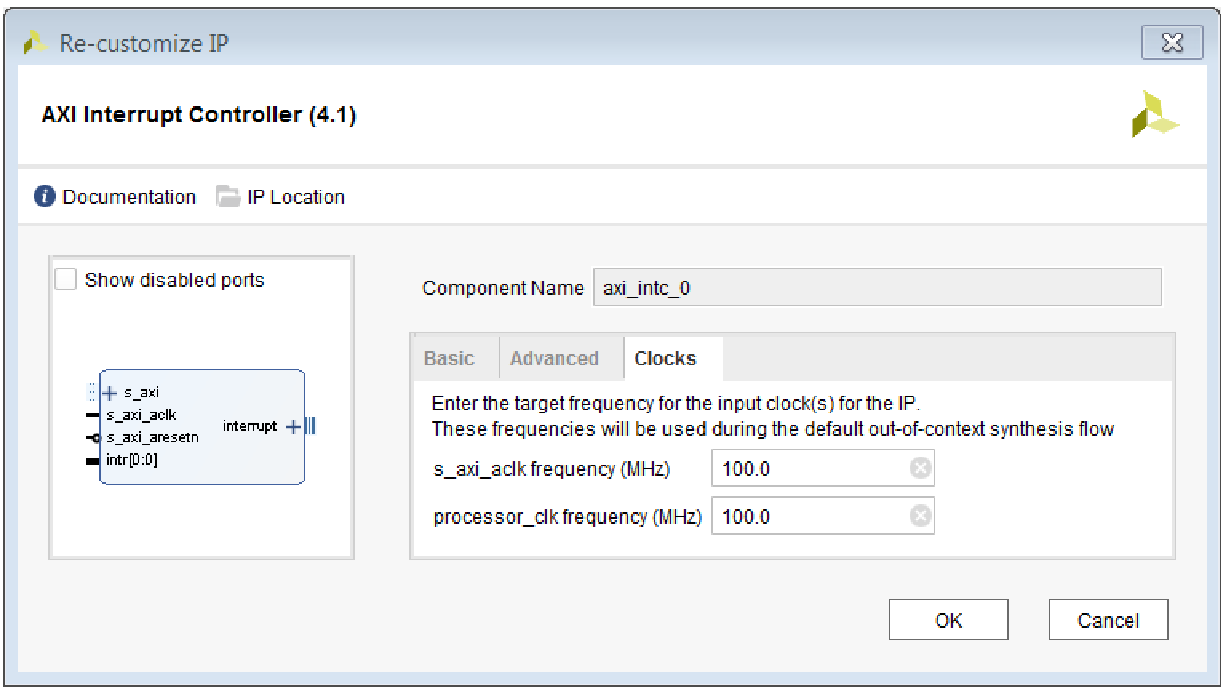
One option of note is the Asynchronous Clocks option. The AXI Interrupt Controller determines whether the interrupt sources in a design are from the same clock domain or different clock domains.

In the case of interrupts being driven from different clock domains, the Vivado IDE uses the Enable Asynchronous Clock operation automatically. In this case, cascading synchronizing registers are added to the interrupt sources.

TIP: *You can also override the automatic behavior by toggling the Auto button to Manual and setting this option manually.*

The Clocks tab lets you specify the Clock Frequencies so constraints can be generated for the Out-of-context (OOC) synthesis flow.

Figure 42: **Interrupt Controller Clocks Tab**



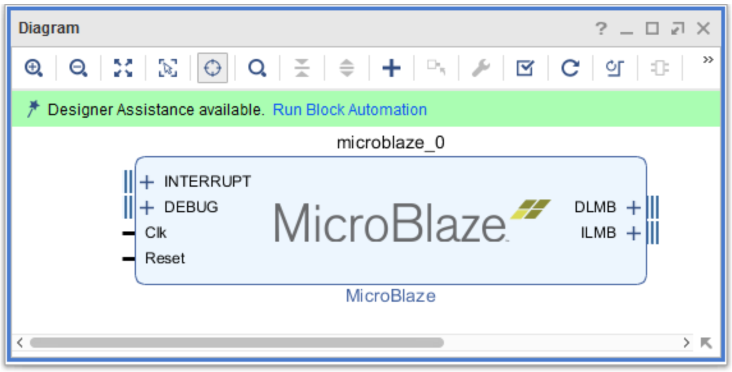
Using the Designer Assistance Feature

IP integrator offers a feature called Designer Assistance, which includes *Block Automation* and *Connection Automation*, to assist you in putting together a basic IP sub-system by making internal connections between different blocks and making connections to external interfaces. The Block Automation Feature is provided when an embedded processor such as the Zynq-7000 Processor System 7 (ZYNQPS7), a Zynq MPSoC (Zynq_ultra_ps_e_0), a MicroBlaze processor, or some other hierarchical IP such as an Ethernet is instantiated in the IP integrator BD.

Using Block Automation

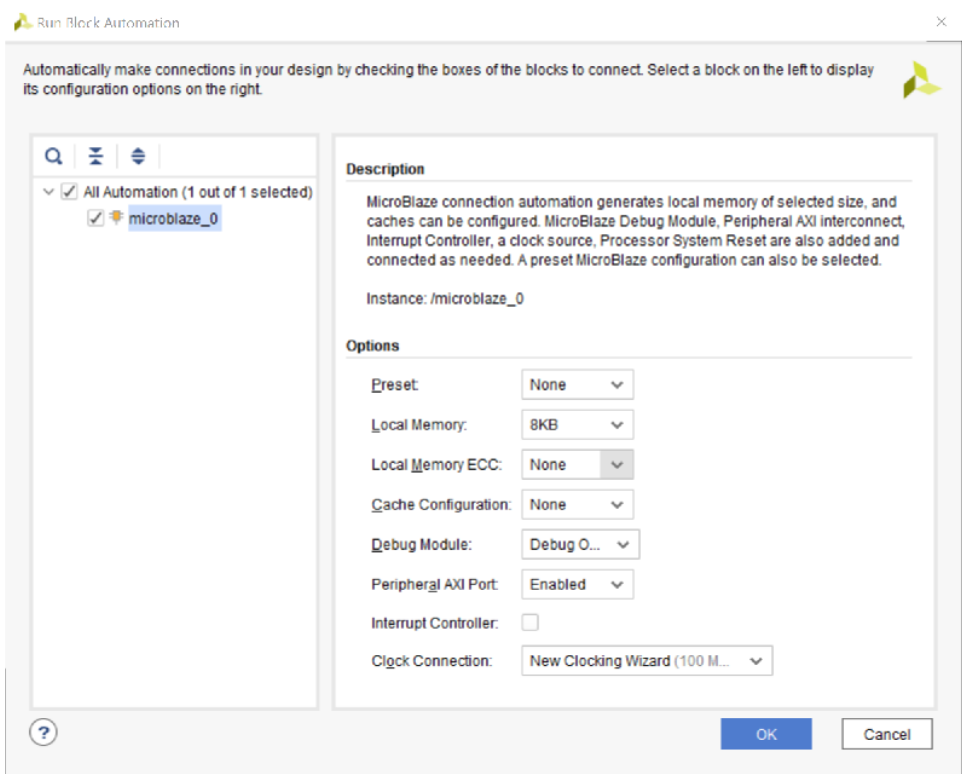
Designer assistance can help you put together a simple MicroBlaze system. To use this feature:

1. Click the **Run Block Automation** link in the banner of the design canvas, as shown in the following figure.

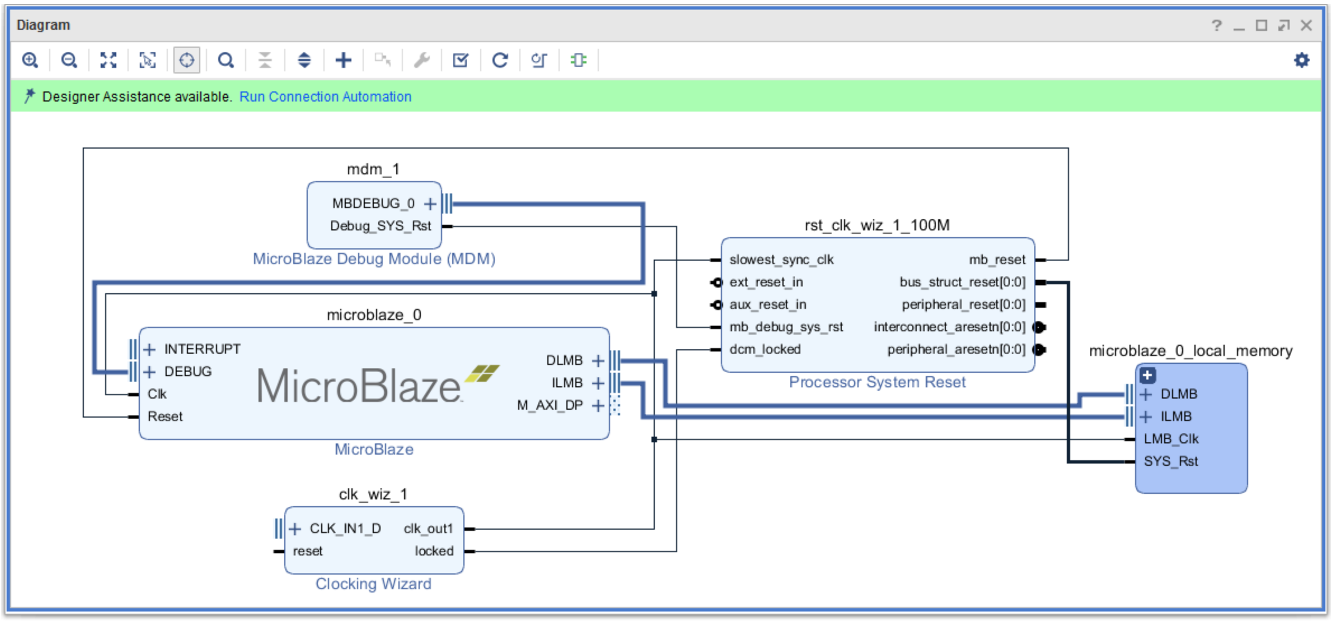


The Run Block Automation dialog box opens, as shown in the following figure.

2. Provide input about basic features that the microprocessor system needs.



After you specify the necessary options, the Block Automation feature automatically creates a basic system, as shown in the following figure.



For example, the MicroBlaze System shown in the following figure consists of the following:

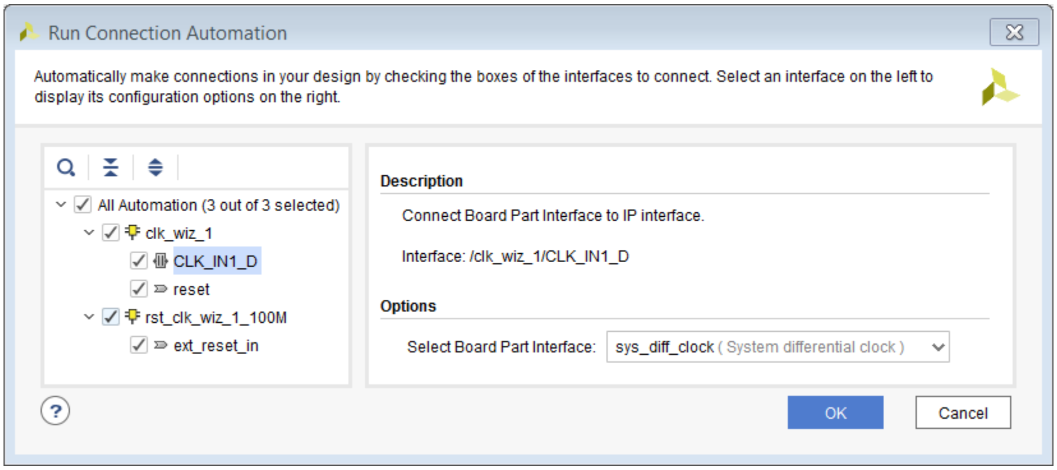
- A MicroBlaze Debug Module
- A hierarchical block called the `microblaze_1_local_memory` that has the Local Memory Bus, the Local Memory Bus Controller and the Block Memory Generator
- A Clocking Wizard
- An AXI Interconnect
- An AXI Interrupt Controller

Using Connection Automation

Because the design is not connected to any external I/O at this point, IP integrator offers the Connection Automation feature as shown in the light green banner of the design canvas in the preceding figure. When you click Run Connection Automation, IP integrator provides assistance in connecting interfaces and/or ports to external I/O ports.

The Run Connection Automation dialog box, shown in the following figure, lists the ports and interfaces that the Connection Automation feature supports, along with a brief description of the available automation, and available options for each automation.

Figure 43: **Ports and Interfaces That Can Use Connection Automation**



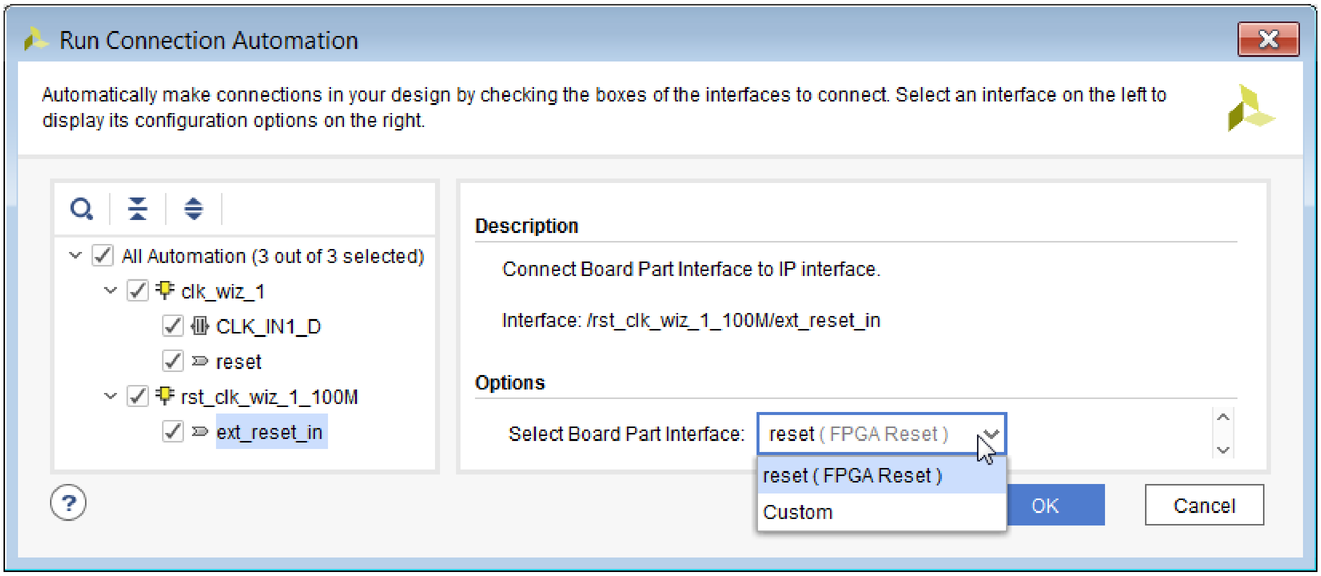
For Xilinx Target Reference Platforms or evaluation boards, IP integrator has knowledge of the FPGA pins that are used on the target boards; this is called *Board Awareness*. Based on that information, the IP integrator connection automation feature can assist you in tying the ports in the design to external ports on the board. IP integrator then creates the appropriate physical constraints and other I/O constraints required for the I/O port in question.

In the MicroBlaze system design shown above, the following connections need to be made:

- Processor System Reset IP needs to be connected to an external reset port.
- Clocking Wizard needs to be connected to an external clock source as well as an external reset.

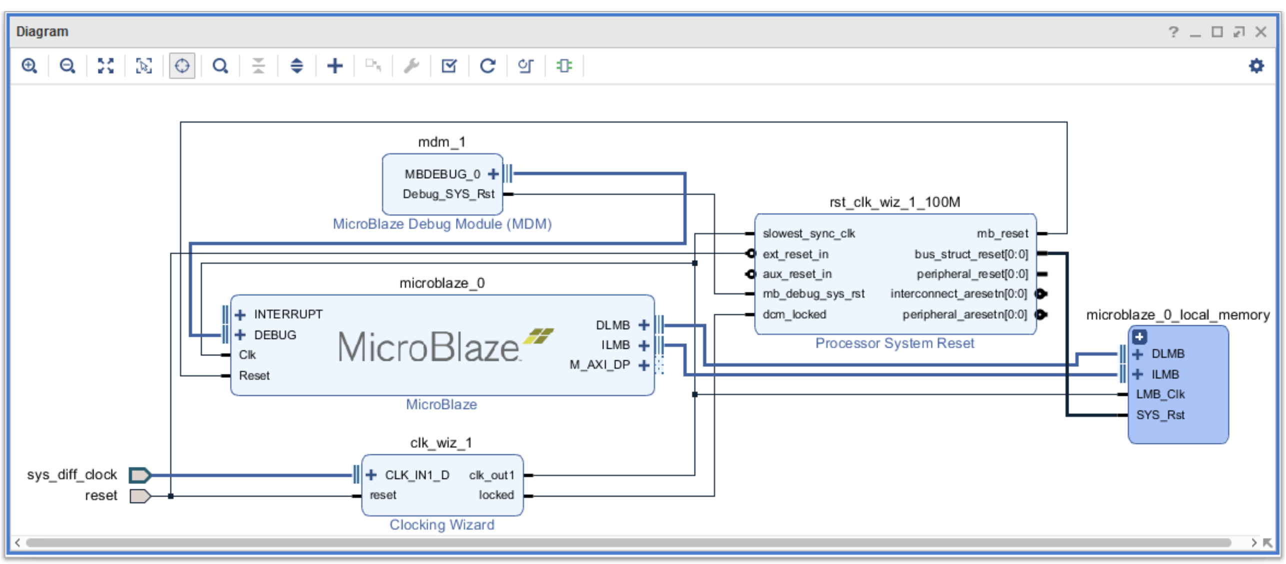
By selecting the appropriate options, as shown in the following figure, you can tie the clock and the reset ports to the appropriate sources on the target board.

Figure 44: **Run Connection Automation Dialog Box to Select Board Interfaces**



You can select the reset pin that already exists on the KC705 target board in this case, or you can specify a custom reset pin for your design. After the reset is specified, the reset pin is tied to the `ext_reset_in` pin of the `Proc_Sys_Rst` IP and the clock is connected to the on-board 200 MHz clock source called `sys_diff_clock`.

Figure 45: **Connecting the Reset Pin to the Board Reset Pin**

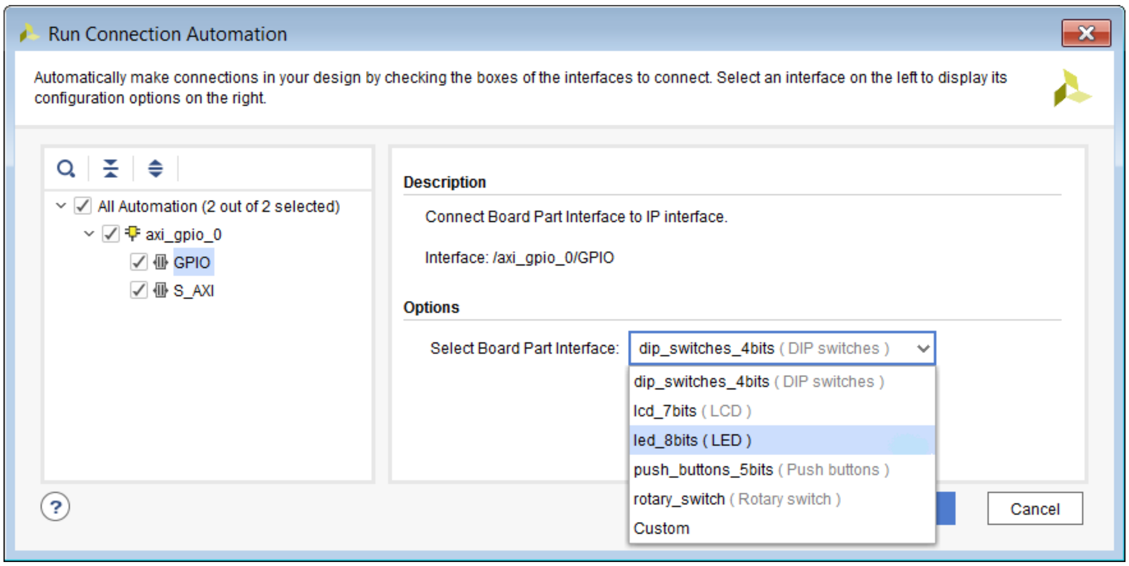


The Designer Assistance feature is constantly monitoring your design development in IP integrator.

For example, assume that you instantiate the `AXI_GPIO` IP into the design. The Run Connection Automation link reappears in the banner on top of the design canvas. You can then click Run Connection Automation and the `S_AXI` port of the newly added AXI GPIO can be connected to the MicroBlaze processor using the AXI Interconnect.

Likewise, the GPIO interface can be tied to one of the several interfaces present on the target board. (See the following figure.)

Figure 46: **Using Connection Automation to Show Potential Connections**



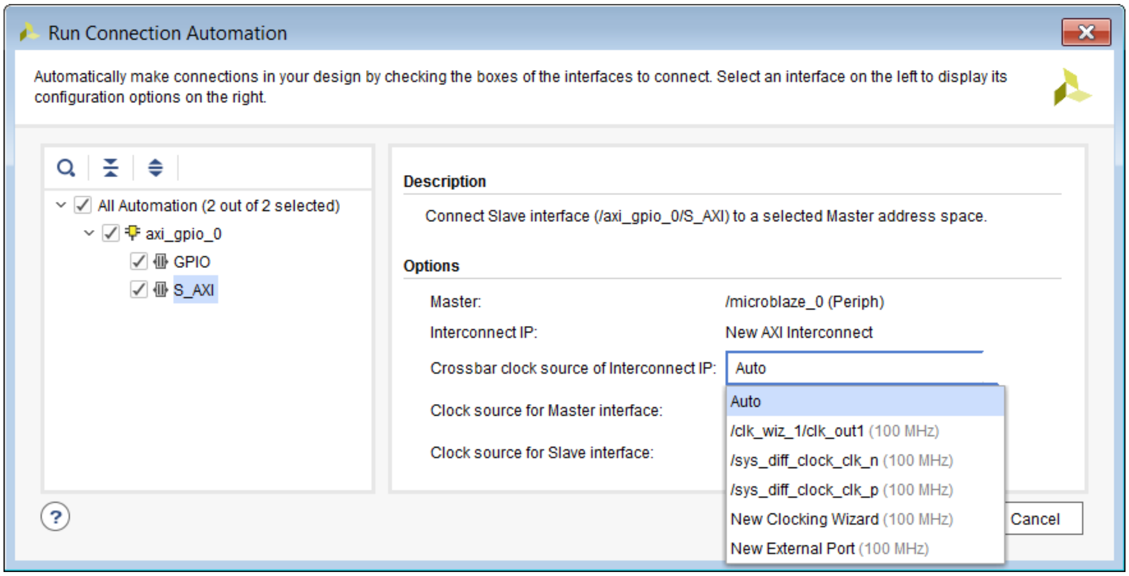
The connection options are as follows:

- The GPIO interface port can be connected to either the Dip Switches that are 4-bits, or to the LCD that are 7-bit or 8-bit, or the 5-bits of Push Buttons.
- The Rotary Switch on the board can be connected to a Custom interface.

Selecting any one of the choices connects the GPIO port to the existing connections on the board.

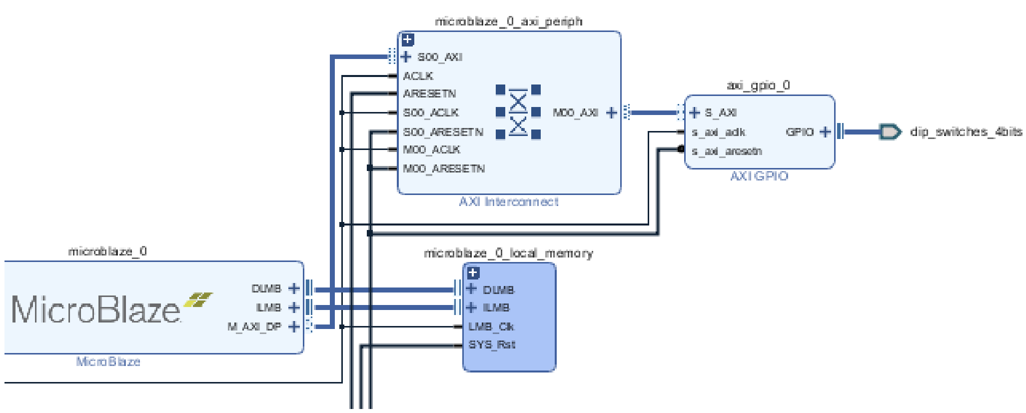
Selecting the `S_AXI` interface for automation, as shown in the following figure, informs you that the slave AXI port of the GPIO can be connected to the MicroBlaze master. If there are multiple masters in the design, then you have a choice to select between different masters. You can also specify the clock connection for the slave interface such as `S_AXI` interface of the GPIO.

Figure 47: **Connecting the Slave Interface S_AXI to the MicroBlaze Master**



When you click the OK in the Run Connection Automation dialog box, the connections are made and highlighted as shown in the following figure.

Figure 48: **Master/Slave Connections**



Using Enhanced Designer Assistance

Enhanced Designer Assistance is available for advanced users who want to connect an AXI4-Stream interface to a memory-mapped interface. In this case IP integrator instantiates the necessary sub-components and makes appropriate connections between them to implement this functionality. See this link in the *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898) for more information on this feature.

Using the Signals View to Make Connections

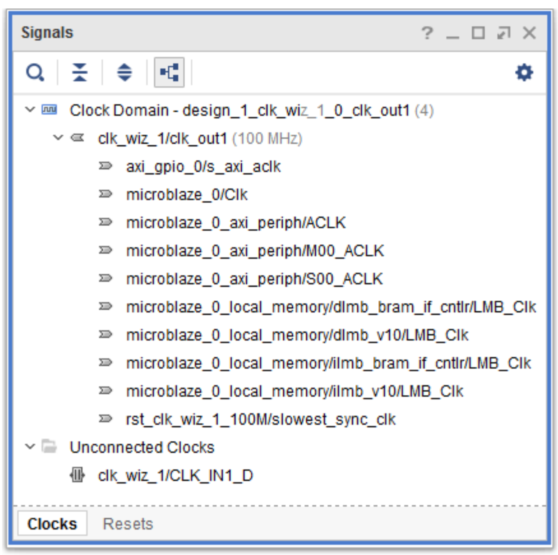
After a BD is open, the Signals window displays, as shown in the following figure, with two tabs listing the Clocks and Resets present in the design.

Selecting the appropriate tab displays the clock or reset signals in the design, and provides an easy way to make connections to the signals.

Clocks are listed in the Clocks view based on the clock domain name. In the following figure, the clock domain is `design_1_clk_wiz_1_0_clk_out1` and the output clock is called `clk_out1` with a frequency of 100 MHz, and is driving several clock inputs of different IP.

When you select a clock from the Unconnected Clocks folder, IP integrator highlights the respective clock port in the BD. Right-clicking the selected clock presents you with several options.

Figure 49: **Signals Window**

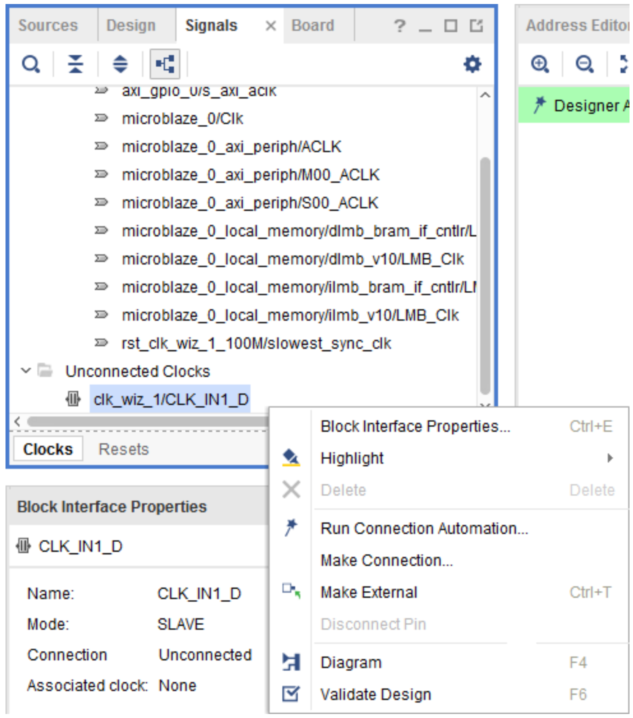


In the MicroBlaze design case shown above, the Designer Assistance is in the form of the Run Connection Automation command that you can use to connect the `CLK_IN1_D` input interface of the Clocking Wizard to the clock pins on the board.

You can also select the **Make Connection** command, and connect the input to an existing clock source in the design. Finally, you can tie the pin to an external port by selecting the **Make External** command.

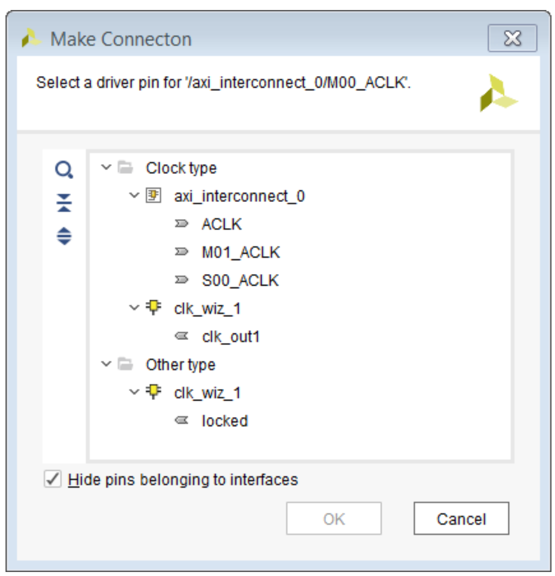
Other options for switching the context to the diagram and running design validation are also available.

Figure 50: **Making Connection using the Signals Window**



When you select Make Connection, a dialog box opens, if a valid connection can be made.

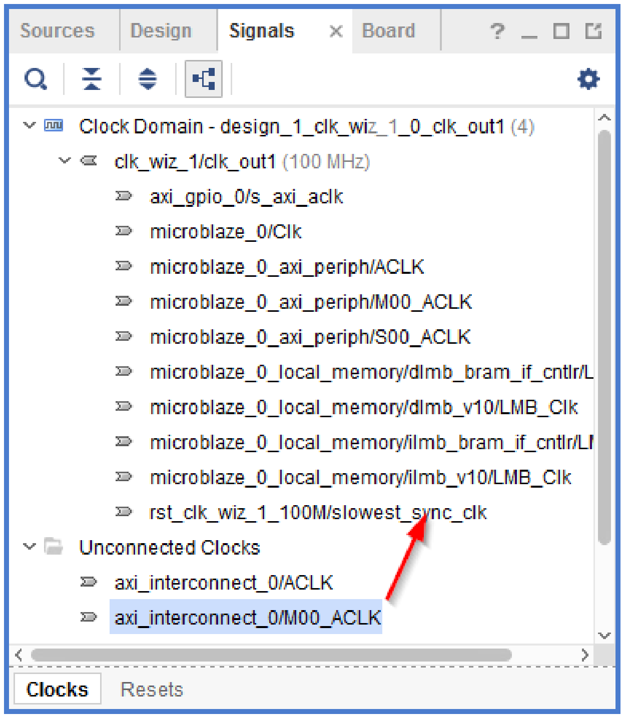
Figure 51: **Make Connection Dialog Box**



Selecting the appropriate clock source makes the connection between the clock source and the port or pin.

If there are unconnected clock pins on one or more cells in the BD, they list in the Unconnected Clocks folder of the Signals window. You can select an unconnected clock pin and drag and drop it to a desired clock domain.

Figure 52: **Drag and Drop an Unconnected Clock into an Existing Clock**

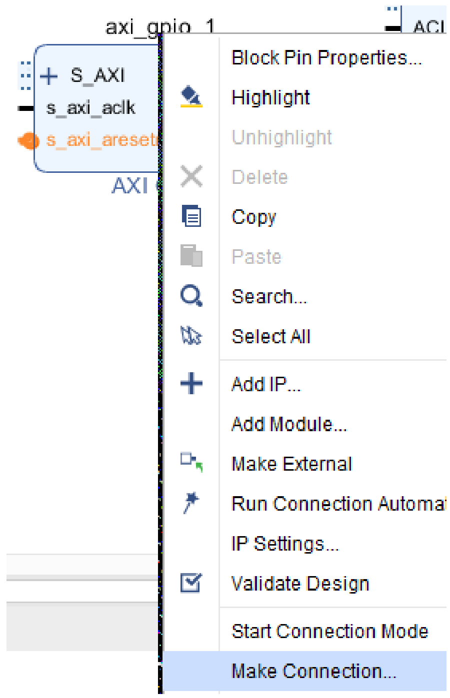


Connections can similarly be made from the Resets tab. Using the Clocks and Resets views of the Signals window provides you with a visual way to manage and connect clocks and resets in the design.

Using Make Connections to Connect Ports and Pins

Connections to unconnected ports or pins can be made by selecting a port or pin and then selecting Make Connection from the right-click menu, as shown in the following figure.

Figure 53: **Make Connection Command**

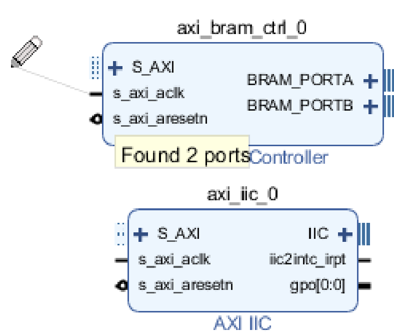


If a valid connection to the selected pin exists, the Make Connection dialog box opens to show all the possible sources to which that the net can be connected. From this dialog box you can select the appropriate source to drive the port or pin.

Making Connections with Start Connection Mode

You can quickly make connections by clicking on a pin of an IP or module and, when the pencil icon is displayed, dragging the cursor to another pin and releasing the mouse as shown in the following figure.

Figure 54: **Starting Connection Mode to Make Connections**



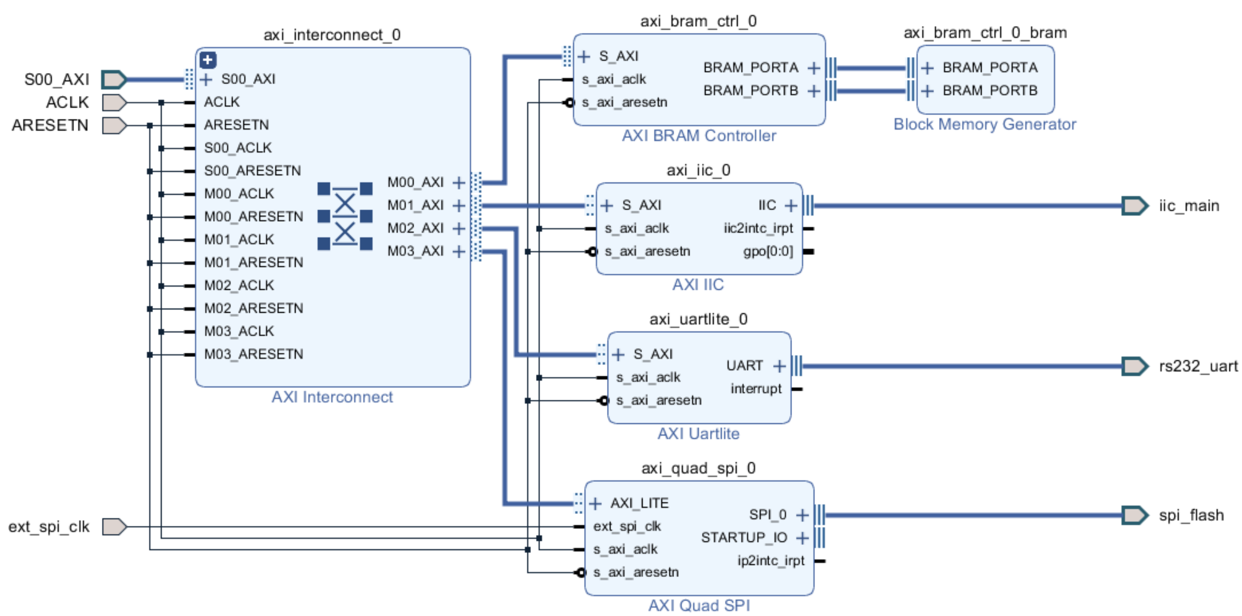
After the connection is made to the `s_axi_aclk` pin of the AXI BRAM Controller, the Start Connection mode will offer to connect the signal to the `s_axi_aclk` pin of AXI IIC, or any other adjacent compatible pins. In this way connections from a source pin can quickly be made to multiple different load pins.

Interfacing with AXI IP Outside of the Block Design

There are situations when the AXI master is outside of the BD and connecting to AXI slaves inside the design. These external masters are typically connected to the BD using an AXI Interconnect. After the ports on the AXI interconnect are connected to an external port, by the Create Interface Port or Make External commands, the address editor is available in the IP integrator and memory mapping can be done as described in Chapter 3: Creating a Memory Map.

As an example, consider the BD shown in the following figure.

Figure 55: **Example Design with External AXI Master Interfacing with Block Design**



When the `S00_AXI` interface of the Interconnect is made external, the Address Editor window becomes available, and memory mapping all the slaves in the BD can be done in the normal manner.

Re-Arranging the Design Canvas

You can re-arrange IP blocks on the canvas to get a better layout of the BD, and connections between blocks. To arrange a completed diagram or a diagram in progress, you can click the Regenerate Layout button.

You can also move blocks manually by clicking a block, holding the left-mouse button down, and moving the block with the mouse, or with the arrow keys.

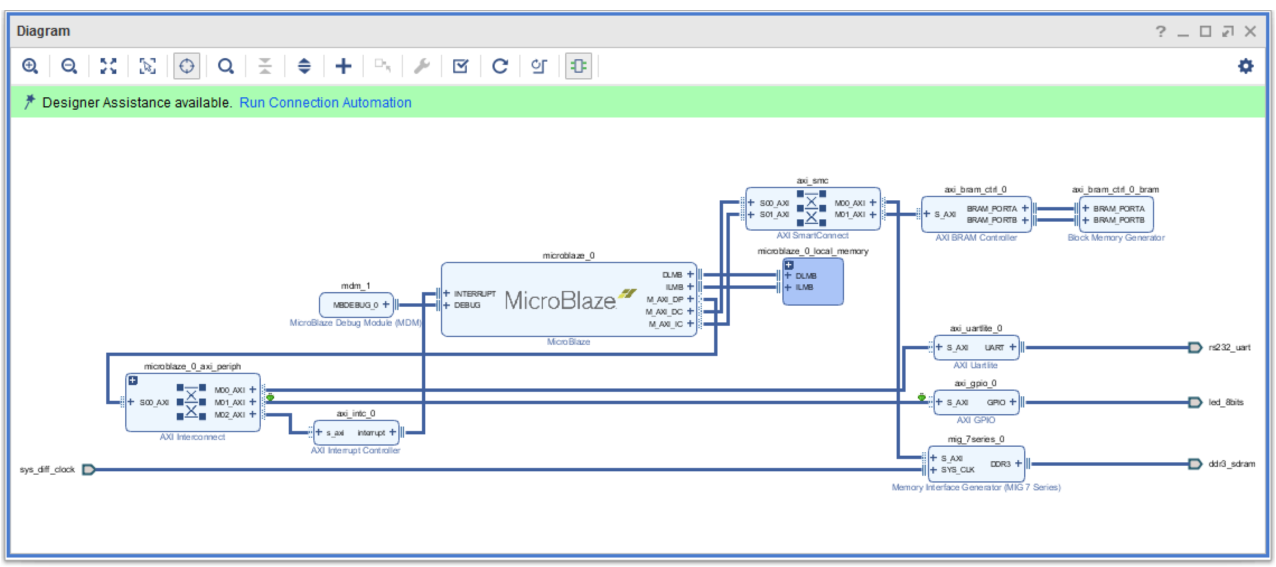
The diagram only allows specific column locations, indicated by the dark gray vertical bars that appear when moving a block. A grid appears on the diagram when moving blocks, which assists you in making better block and pin alignments.

It is also possible to manually place the blocks where desired, and then click Optimize Routing. This command preserves the placement of the blocks, unlike the Regenerate Layout command, and only modifies the routing to the placed blocks.

Showing Interface Level Connectivity Only

To see only the connectivity between interfaces present on the BD select the Show interface connections only button from the BD toolbar. As seen in the following figure, this shows only the interface level connections, and hides all the other connections.

Figure 56: **Interface Connections Only**

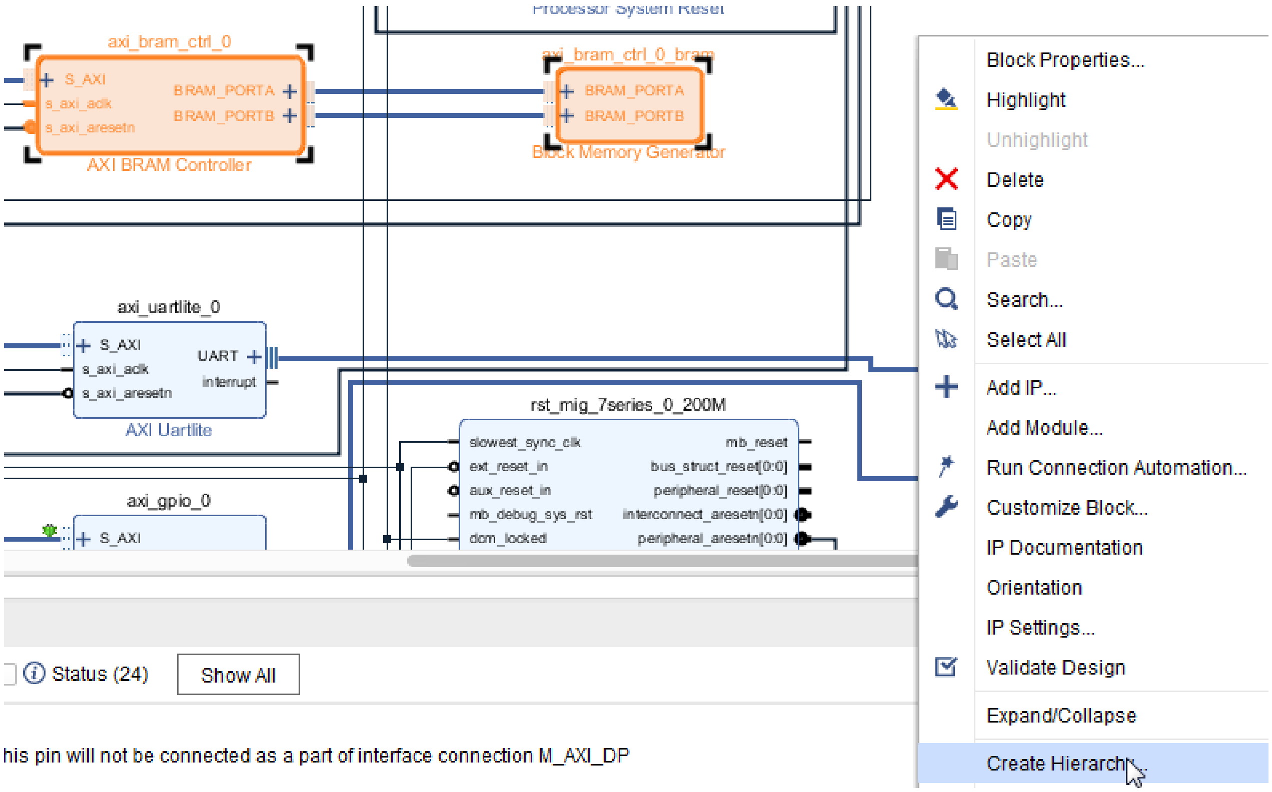


Clicking the Show interface connections only button again restores all the connections in the BD.

Creating Hierarchies

You can create a hierarchical block in a diagram by using Ctrl+Click to select the desired IP blocks, right-click and select **Create Hierarchy**, as shown in the following figure. The IP integrator creates a new level of hierarchy containing the selected blocks.

Figure 57: **Create Hierarchy**

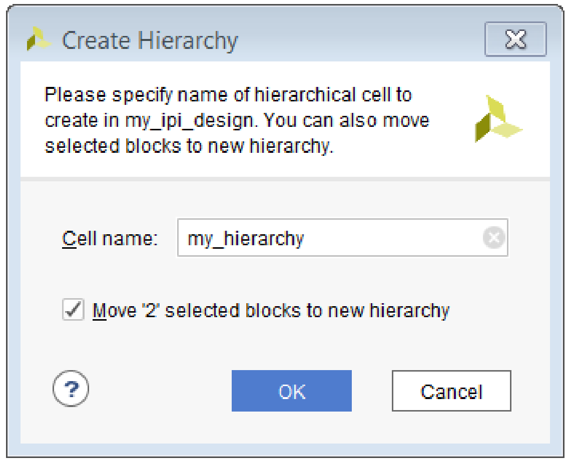


Creating multiple levels of hierarchy is supported. You can also create an empty level of hierarchy, and later drag existing IP blocks into that empty hierarchical block.

When you click the + sign in the upper-left corner of an expandable block you can expand the hierarchy. You can traverse levels of hierarchy in a diagram using the Explorer type path information displayed in the upper-left corner of the IP integrator diagram.

Clicking Create Hierarchy opens the Create Hierarchy dialog box, as shown in the following figure, where you can specify the name of the new hierarchy.

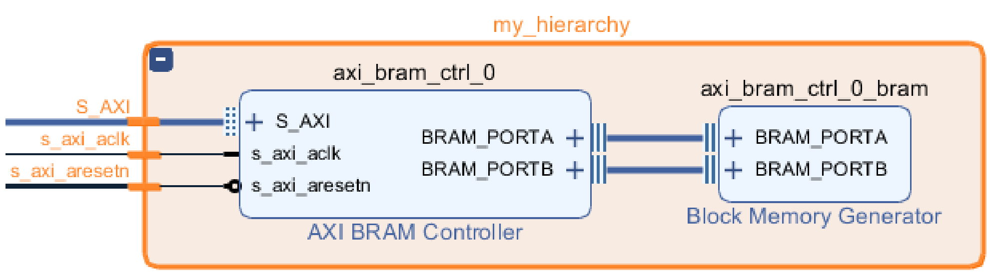
Figure 58: **Create Hierarchy Dialog Box**



This action groups the selected IP blocks under one block, as shown in the following figure.

- Click the + sign of the hierarchy to view the components underneath.
- Click the – sign on the expanded hierarchy to collapse it back to the grouped form.

Figure 59: **Cells Under Hierarchical Block**

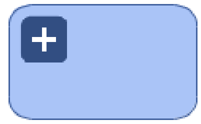


Adding Pins and Interfaces to Hierarchies

As mentioned above, you can create an empty hierarchy and you can define the pin interface on that hierarchy before moving blocks of IP under the hierarchy.

Right-click the IP integrator canvas, with no IP blocks selected, and select **Create Hierarchy**. In the Create Hierarchy dialog box, you specify the name of the hierarchy. After the empty hierarchy is created, the BD should look like the following figure.

Figure 60: **Empty Hierarchy**



You can add pins to this hierarchy by typing the create_bd_pin command at the Tcl Console:

```
create_bd_pin -dir I -type rst /hier_0/rst
```

In the above command, an input pin named `rst` of type rst was added to the hierarchy. You can add other pins using similar commands. Likewise, you can add a clock pin to the hierarchy using the following Tcl command:

```
create_bd_pin -dir I -type clk /hier_0/clock
```

You can also add interfaces to a hierarchy by using the following Tcl commands. First set the BD instance to the appropriate hierarchy where the interface is to be added, using the current_bd_instance Tcl command:

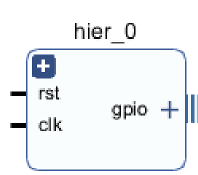
```
current_bd_instance /hier_0
```

Next, create the interface using the create_bd_intf_pin Tcl command as follows:

```
create_bd_intf_pin -mode Master -vlnv xilinx.com:interface:gpio_rtl:1.0 gpio
```

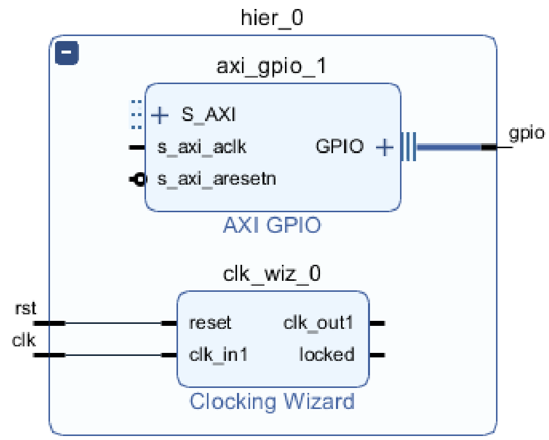
It is assumed that the right type of interface has been created prior to using the above command. After executing the commands shown above the hierarchy should look as shown in the following figure.

Figure 61: **Create Pins**



After you have created the appropriate pin interfaces, different blocks can be dropped within this hierarchical block and pin connections from those IP to the external pin interface can be made.

Figure 62: **Connected IP to Hierarchical Pin Interface**



Cutting and Pasting

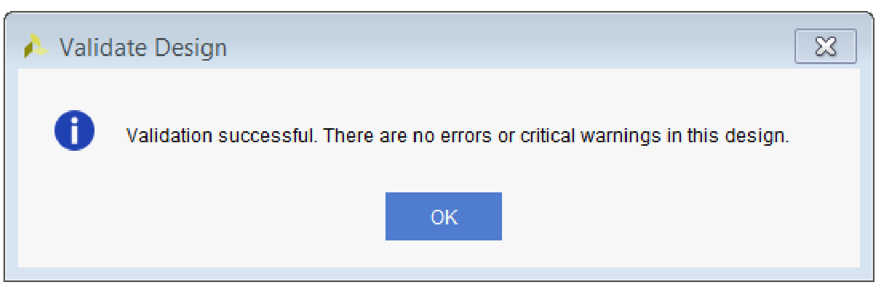
You can use Ctrl+C and Ctrl+V to copy and paste blocks in a diagram. This lets you quickly copy IP blocks that have been customized, or copy IP into new hierarchical blocks.

Running Design Rule Checks

IP integrator runs basic design rule checks in real time as the design is being assembled. However, there is a potential for something to go wrong during design creation. As an example, the frequency on a clock pin may not be set right. As shown in the following figure, you can run a comprehensive design check on the design by clicking the Validate Design button  in the toolbar on the IP integrator canvas.

If the design is free of Warnings and/or Errors, a confirmation dialog box displays, as shown in the following figure.

Figure 63: **Validation Successful Message**



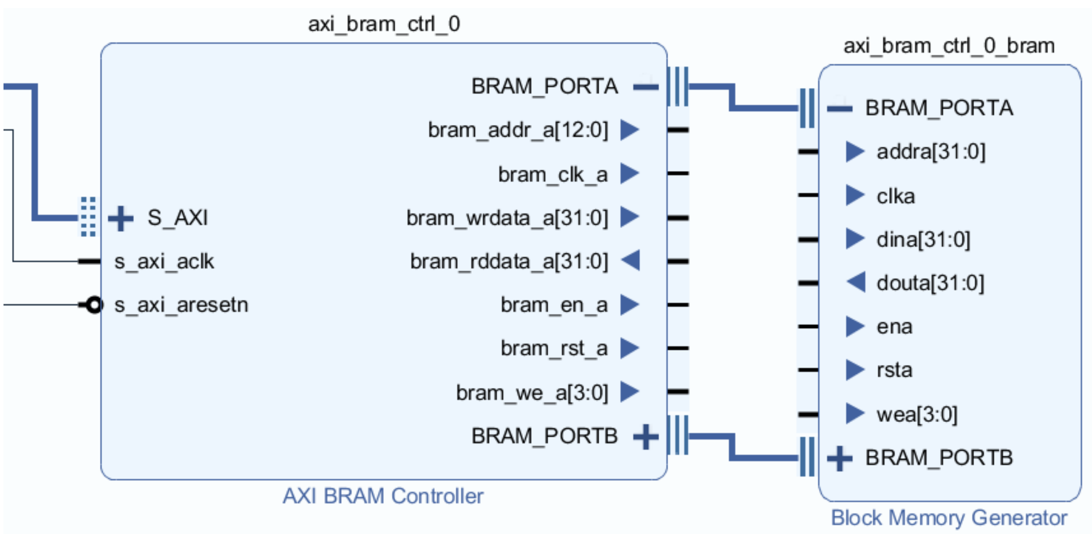
Connecting Ports with Different Widths

It is permitted to connect ports or pins with different widths in IP integrator.

As seen in the following figure, the `bram_addr_a` pin of the AXI BRAM controller that is 14-bits wide is connected to the `addra` pin of the Block Memory Generator that is 32-bits wide. The port width mismatch is not flagged during design validation; however, a warning is issued during the generation of the lock design output products, as follows:

```
[BD 41-235] Width mismatch when connecting pin: '/axi_bram_ctrl_0_bram/
addra'(32) to
net 'axi_bram_ctrl_0_BRAM_PORTA_ADDR'(14) - Only lower order bits will be
connected.
```

Figure 64: **Connecting Pins of Differing Widths**



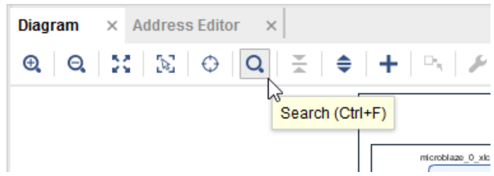
The warning indicates that the tool has detected a port width mismatch while connecting the ports or pins, and that only the lower-order bits (the first 14 bits) will be connected.

You will need to evaluate the warning and take appropriate action as needed. Typically, it is okay to ignore this warning message.

Finding Objects in a Block Design

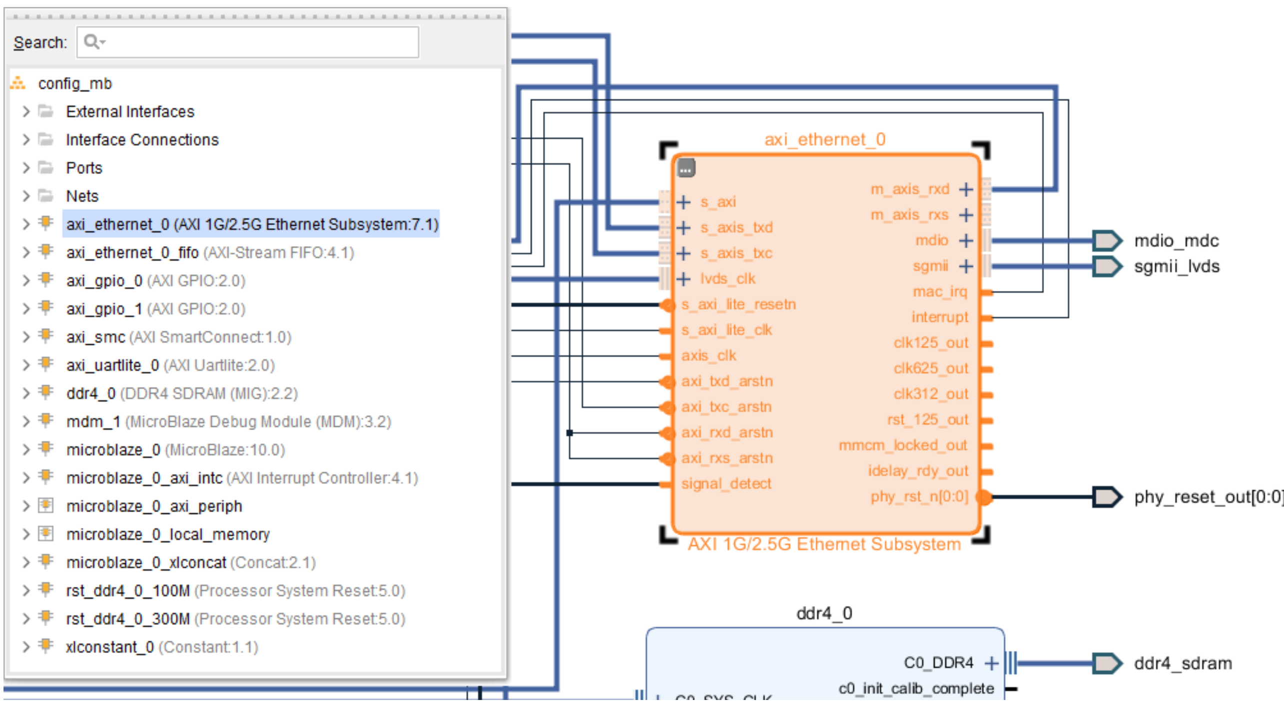
There are two ways of finding objects in a block design. The first one is used in the block design canvas. This functionality can be invoked by clicking the magnifying glass icon on the block design canvas toolbar or by pressing the Ctrl+F keys together.

Figure 65: **Search Function in Block Design Toolbar**



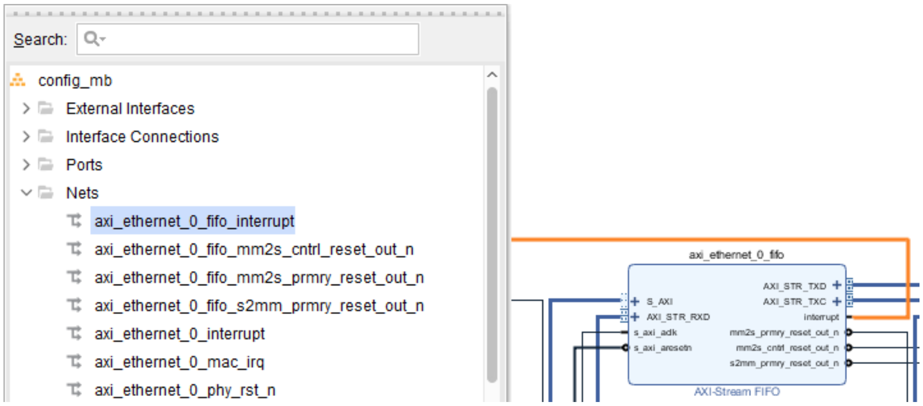
This brings up a Search dialog box that shows the Interfaces, Nets, Ports and Cells in the block design. Selecting an object in the search window highlights the corresponding object in the block design canvas.

Figure 66: **Finding a Cell in the Block Design**



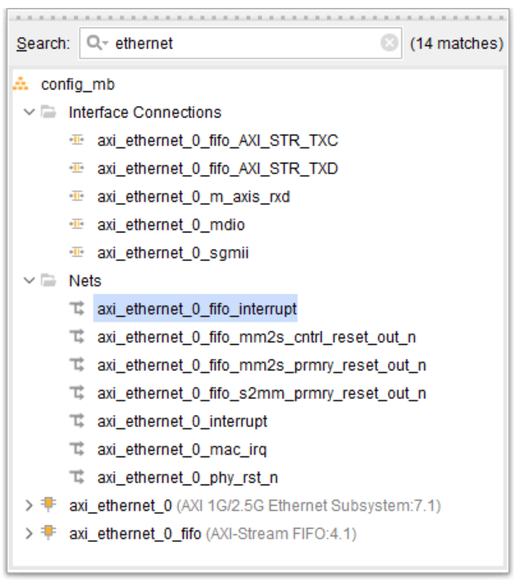
You can expand the Ports, Nets, or Interfaces groupings and select any net to cross-probe.

Figure 67: **Finding a Net in the Block Design**



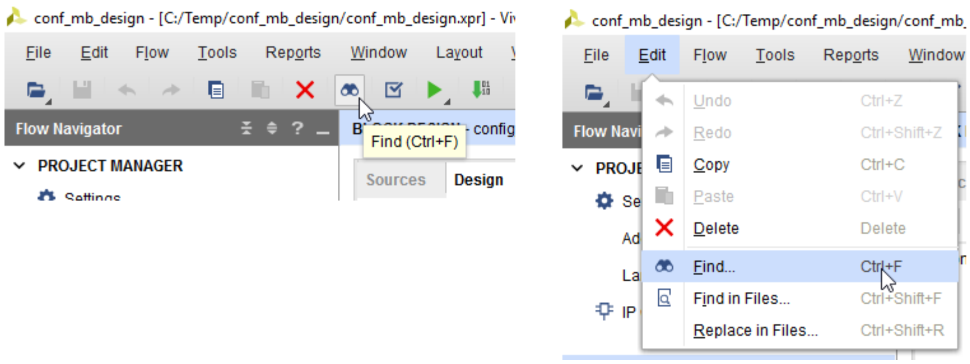
Finally, the Search field in the dialog box allows for searching and filtering for specific objects. As an example, typing ethernet in the Search field shows all the objects matching with ethernet.

Figure 68: **Filtering Objects to Find in a Block Design**



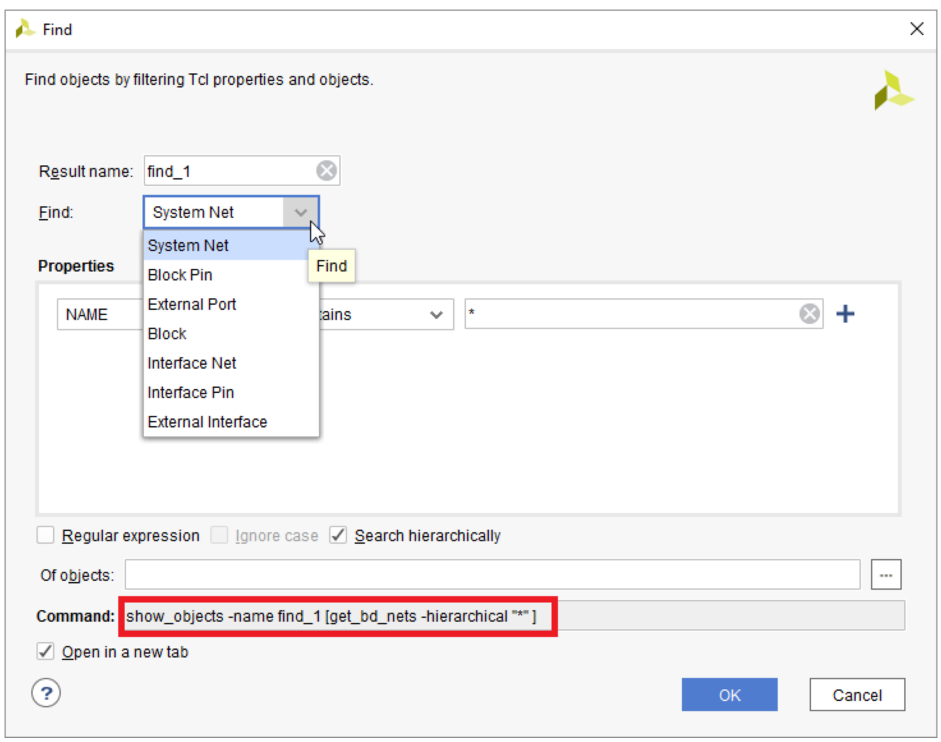
In addition to the Search dialog box described above, there is a Find dialog box that you can open from the Vivado toolbar by clicking on the binocular icon, or by selecting **Edit→Find** from the menu.

Figure 69: **Finding Objects from Vivado Toolbar or Menu**



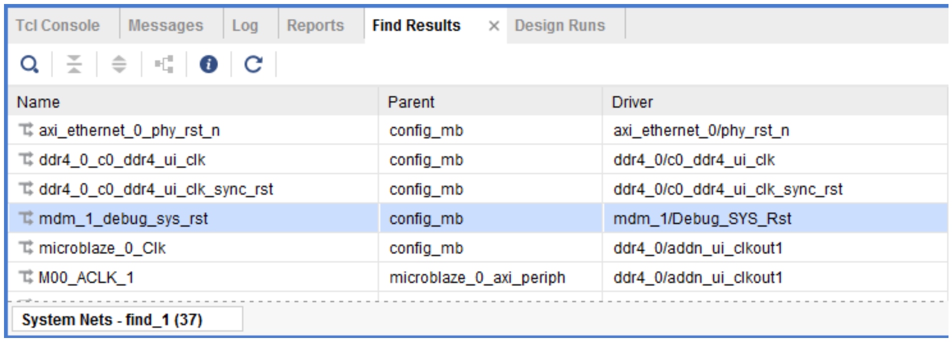
The purpose of this Find dialog box is to work with Tcl commands such as `get_bd_cells`, `get_bd_intf_nets`, `get_bd_intf_pins`, `get_bd_intf_ports`, `get_bd_nets`, `get_bd_pins` and `get_bd_ports`. By invoking the Find dialog box, the user can then search on different kind of block design objects.

Figure 70: **Find Dialog Box**



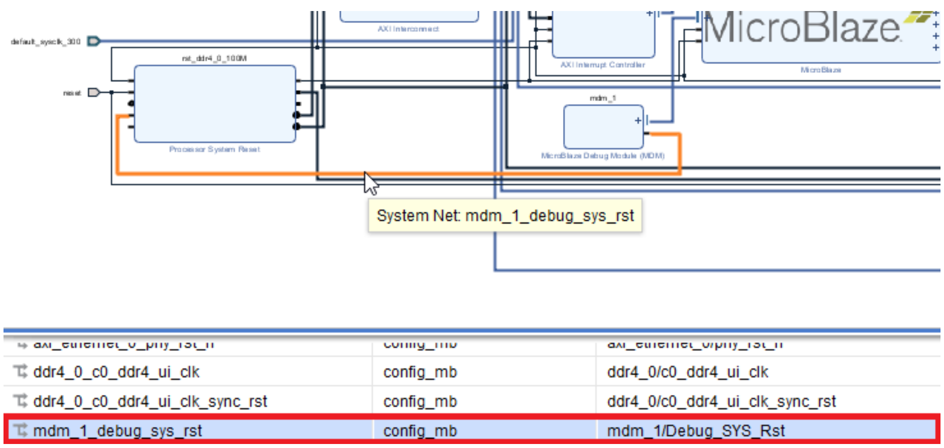
Clicking OK on the dialog box brings up the Find Results window as shown.

Figure 71: **Find Results Window**



Selecting an object within this window cross-probes the corresponding object in the block design canvas.

Figure 72: **Find Results Window and Cross-Probing Objects**



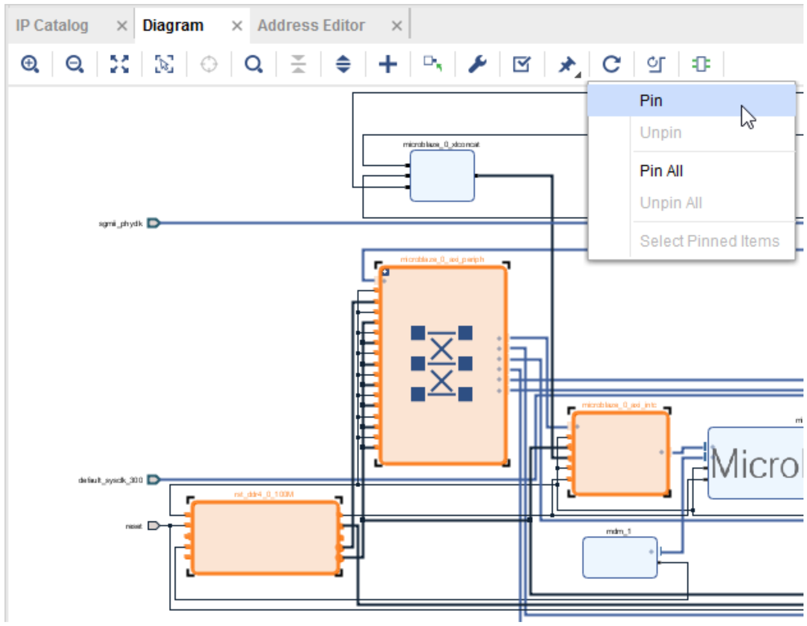
The appropriate Tcl command also shows up on the Tcl Console.

```
show_objects -name find_1 [get_bd_nets -hierarchical "*" ]
```

Pinning Blocks and Ports to Location

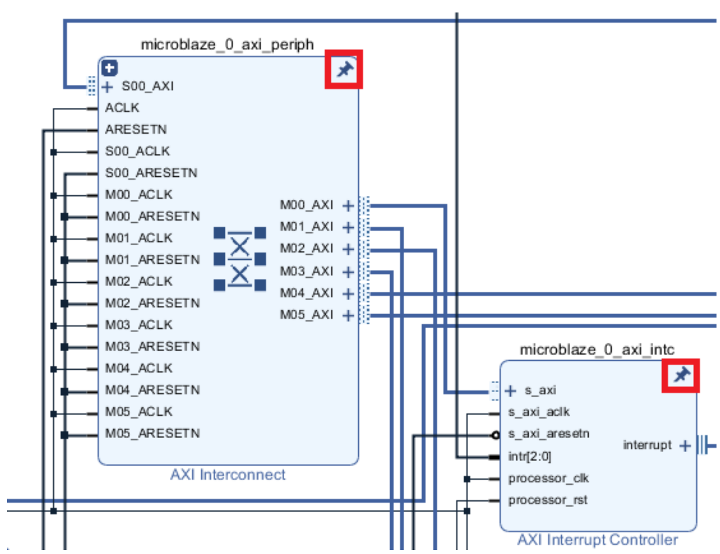
Often times cells in block designs need to arranged in a specific way to show control or data paths of a design. The pinning function in IP integrator provides designers with the ability to “lock” cells with respect to each other and to a particular coordinate on the block design.

Figure 73: **Pinning Blocks in Block Design**



To pin blocks on to a certain location on the canvas, just select one or multiple blocks on the canvas, click the Pin icon on the tool bar and select Pin from the context menu. The pinned blocks are shown with a Pin icon on the top right corner of the cell as shown below.

Figure 74: **Pin Symbol on Cells in the Block Design**



Regenerating the layout of the block design or regenerating an optimal routing will not affect the location of the pinned cells in question. External Ports can also be pinned using the same functionality. Pinning can also be done alternatively, by selecting one or multiple cells (or ports) on the block design canvas, right-clicking and selecting from the context menu **Pinning → Pin**. To unpin an object, select the cell/port on the block design and either by clicking on the Pin icon on the tool bar or right-clicking in the block design, select **Pinning → Unpin** from the context menu.

Chapter 3

Creating a Memory Map

A **Memory Map** can be defined for each **slave Bus Interface** and is made up of one or more **Address Block**, **Bank** and **Subspace Map** elements. A **Memory Map** may be accessed through a slave interface. Maps the memory a bus slave contributes to a bus' **Address Space**.

The table below describes common terms used in the context of addressing. It is important to get familiar with these terms as any addressing related message flagged by IP Integrator, use these terms. Understanding these terms will help take appropriate measures to solve any addressing related issues.

Table 2: **Terminology**

Term	Description
Bus Interface	The interface of a IP to a Bus. Blocks are connected by linking the Bus Interfaces together. There are three different classes of Bus Interface: • Master - A master interface is the interface mode that initiates a transaction (like a read or write) on a bus. Master interfaces tend to have associated address spaces (that is, address spaces with programmer's view). • Slave - A slave interface is the interface mode that terminates or consumes a transaction initiated by a master interface. Slave interfaces often contain information about the registers that are accessible through the slave interface. • System - A system interface is neither a master nor slave interface; this interface mode allows specialized (or non-standard) connections to a bus, such as external arbiters. System interfaces can be used to handle situations not covered by the bus specification or deviations from the bus specification standard.
Address Space	An Address Space is defined as a logical addressable space of memory. Each master Bus Interface can be assigned a logical Address Space. Address Spaces are effectively the programmer's view looking out from a master interface. An Address space may be partitioned into one or more Address Segments. A Master Interface may reference a single Address Space only.
Address Block	An Address Block describes a single, contiguous block of memory that is part of a memory map.
Slave Segment	An Address Block with an associated Slave Interface (that is, accessible via Network). A Slave Interface may reference multiple Address Blocks within a single Memory Map.
Master Segment	A Master Segment is an assigned Slave Segment within an Address Space (with associated Master Interface).
Address Offset	This is the offset from the start of an Address Space or Address Block.

Table 2: **Terminology** *(cont'd)*

Term	Description
Address Range	Specifies the total addressable range of an Address Space or Address Block. This is the size of the Address Space. An Address Range must be a power of 2 (i.e. 2^N). Address Offset and Address Range are coupled by Alignment. That is, there must be enough "zero" LSBs in the Address Offset to cover the Address Range.
High Address	This is the last addressable address in a particular assigned Segment. It automatically adjusts itself based on the Offset Address and the Range values.
Addressing Path	
Bridge	A Bridge describes an interconnect across an IP, between a slave interface and a master interface, on separate Buses which may be the same or different types. A Bridge describes the internal point-to-point connections between Bus Interfaces. A Bridge can have multiple Address Spaces, supports memory mapping and remapping, and can only have direct interfaces. The transfer of addressing information from the slave interface to the master interface of a bridge is done through the address space assigned to the master interface. This address space defines the visible address range from this master interface. There are two Bridge types: • A Transparent Bridge where an address is not modified from the slave interface into the Address Space of the master interface. • An Opaque Bridge where an address is modified (for example offset by Base Address) from the slave interface into the Address Space of the master interface. If a Bridge is opaque then the whole of the address range is mapped by the Bridge and there are no gaps.
Aperture	An Aperture is a window within a master Address Space, specified as an Address Offset and Address Range, to view a portion of a Slave Segment. Here a Slave Segment represents a physical memory of a particular peripheral or memory device. By default an Address Space is configured with a single Aperture covering its full Address Range, allowing access to a single Slave Segment. However an Address Space may also split its available Address Range across multiple aligned Apertures allowing us to access multiple Master Segments (one per Aperture). Apertures may not overlap.
Dependent Master	Some masters (for example, DMA IP) are themselves driven by other masters and hence are considered Dependent Masters. There is often a related Slave Interface which is used by masters like processors to control the behavior of a Dependent Master.
Pass-through	In a level of a hierarchy, an external slave segment mapped into an external address space.
Sparse Connectivity	In a multiple master design users might want to specify slaves that could potentially be accessed by all masters or by certain masters only. This feature of memory mapping in IP Integrator is called Sparse Connectivity.
Fixed Address	A Fixed Address means the Slave Segment must always be assigned to an Address Space at a fixed address.
Floating Address	A Floating Address means the Slave Segment can be assigned to an Address Space that can change.

Master interfaces reference an assigned memory range container called *address spaces*, or `bd_address_space` objects. Slave interfaces reference a requested memory range container, called a memory map.

By convention:

- Memory Maps are named after the slave interface pins that reference them, for example the `S_AXI` interface references the `S_AXI` memory map, though that is not required.
- Address space names are related to its usage; for example, the MicroBlaze processor has a Data address space and an Instruction address space.

The memory map for each slave interface pin contains slave segments, or `bd_address_seg` objects. These address segments correspond to the address decode window for that slave.

A typical AXI4-Lite slave has only one address segment, representing a range of memory; however, some slaves like a bridge, have multiple address segments, or a range of addresses for each address decode window.

When a slave segment is mapped to the master address space, the IP integrator creates a master `bd_address_seg` object, mapping the address segments of the slave to the master. The Vivado® IP integrator can automatically assign addresses for all slaves in the design. However, you can also manually assign the addresses using the Address Editor.

TIP: *The Address Editor window opens only if the diagram contains an IP block that functions as a bus master (such as the MicroBlaze processor) or if an external bus master (outside of IP integrator) is present.*

Click the **Address Editor** window above the design canvas. In the Address Editor, you can see the address segments of the slaves, and can map them to address spaces in the masters.

If you generate the RTL from an IP integrator block design (BD) without first generating addresses, the IP integrator prompts you to automatically assign addresses at that point.

You can also set addresses manually by entering values in the Offset Address and Range columns.

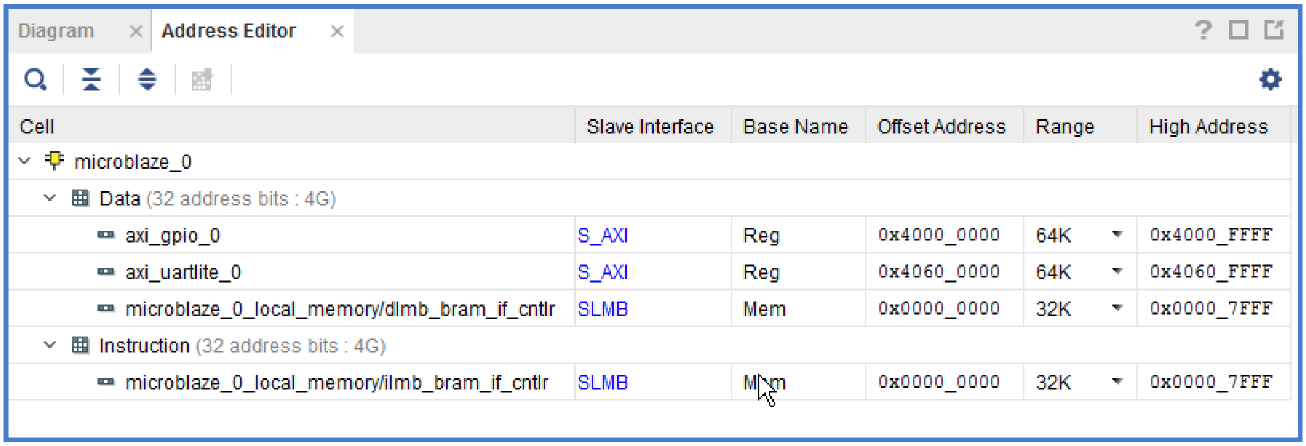
A master, such as a processor, communicates with peripheral devices through device registers. Each of the peripheral devices is allocated a block of memory within an overall memory space of a master. The IP integrator follows the industry standard IP-XACT data format for capturing memory requirements and capabilities of endpoint masters and slaves.

IP integrator provides an Address Editor to allocate these memory ranges to the master/slave interfaces of different peripherals. Master and slave interfaces each reference specific memory objects.

Using the Address Editor

The Address Editor lets you allocate memory ranges to peripherals from the perspective of a master interface. The Address Editor window becomes available when a master with an address space, such as a MicroBlaze processor or a Zynq®-7000 processor is instantiated in the Diagram canvas.

Figure 75: **Address Editor Window**



As the peripherals are instantiated and connected to the processor in the BD canvas using connection automation, the IP integrator automatically enters a corresponding memory assignment to that peripheral in the Address Editor.

The columns of the Address Editor are as follows:

- Cell: Describes the master and the connected peripherals that can be addressed by that master. You can expand the tree by clicking the Expand All button.

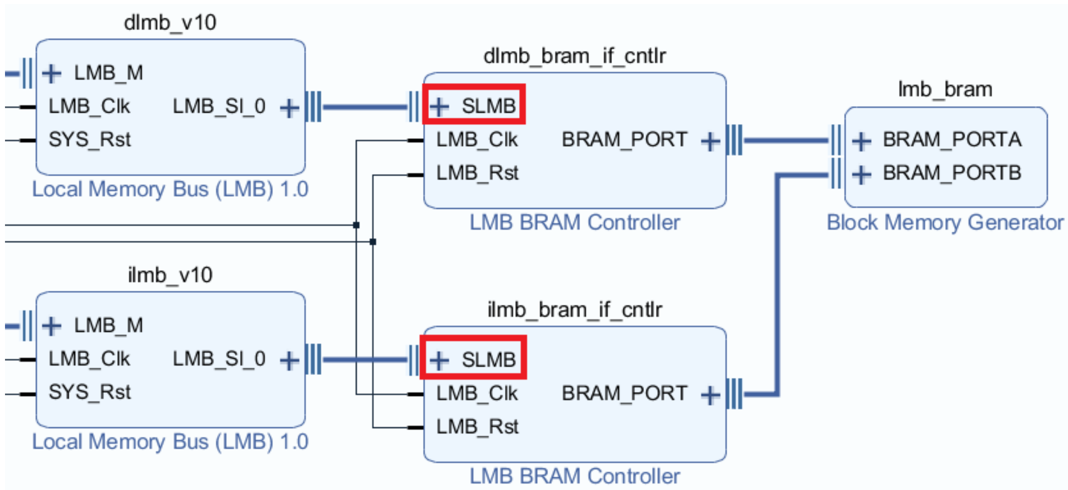
As shown above, the instance name of the “master” is `microblaze_0` which addresses the `Data` and `Instruction` address spaces.

The peripherals `microblaze_0_local_memory/dlmb_bram_if_cntlr` and `microblaze_0_local_memory/ilmb_bram_if_cntlr` are mapped into the `Data` and `Instruction` address spaces respectively, where the rest of the peripheral are only accessible by the `Data` address space.

- Slave Interface: Lists the name of the slave interface pin of the peripheral instance.

As an example, the peripheral instances the `microblaze_0_local_memory/dlmb_bram_if_cntlr` and the `microblaze_0_local_memory/ilmb_bram_if_cntlr` each have an interface called SLMB, as shown in the following figure.

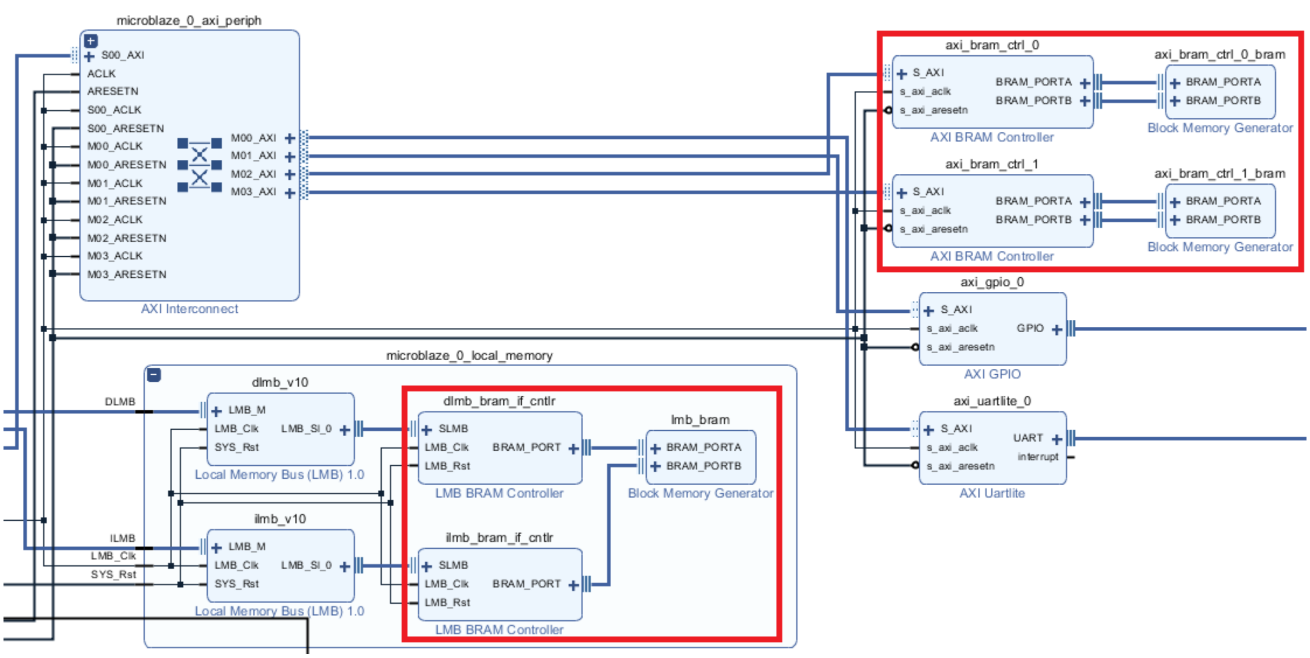
Figure 76: **Interface Names**



- Base Name: Specifies the name of the slave segment.

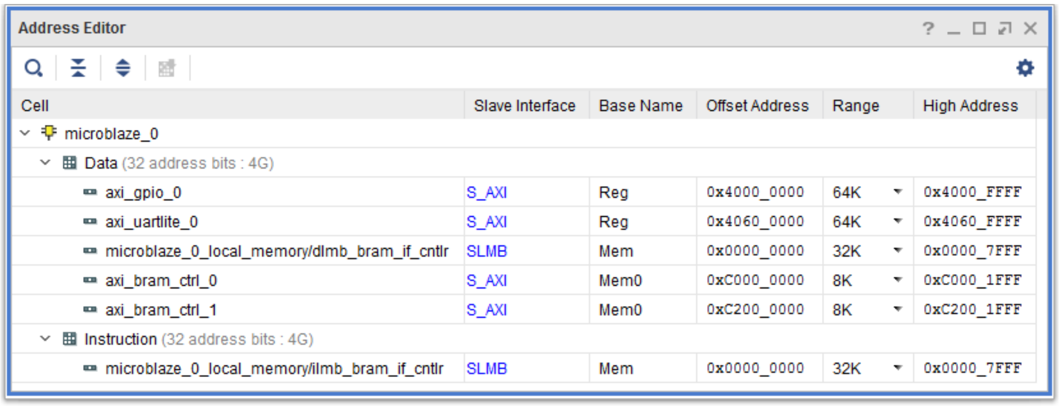
By convention, the two names that IP integrator creates “on-the-fly” are `Mem` (memory) and `Reg` (register), as shown in the following figure, which shows a design with multiple memory instantiations.

Figure 77: **Multiple Memory Instantiations in a Block Design**



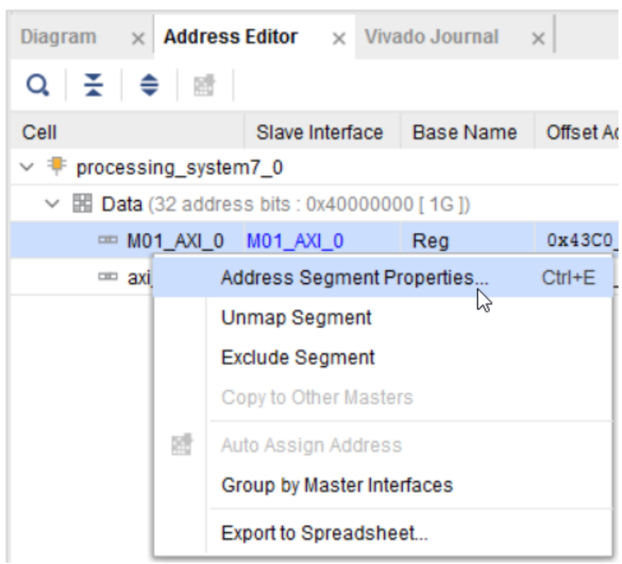
These are given the base names in the address editor as shown in the following figure.

Figure 78: **Base Names given to Multiple Memory Instantiations**



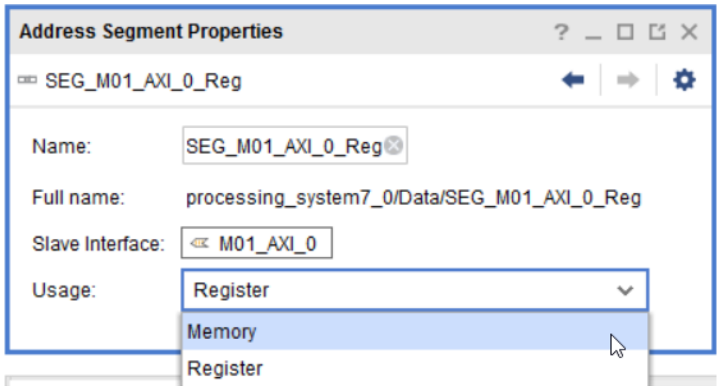
The `Usage` parameter of these memory segments is what defines how the memory segment is to be used. When used as memory the memory segment can be used to run software code, i.e. code and data segments of the application software can be placed into the memory segment. When used as register, they can simply be written to or read from. To change the `Usage` parameter, select the interface, right-click and from the context menu select Address Segment Properties.

Figure 79: **Viewing/Changing Address Segment Properties**



In the Address Segment Properties window, select the right value for Usage from the pull-down menu.

Figure 80: **Setting Usage in Address Segment Properties Window**



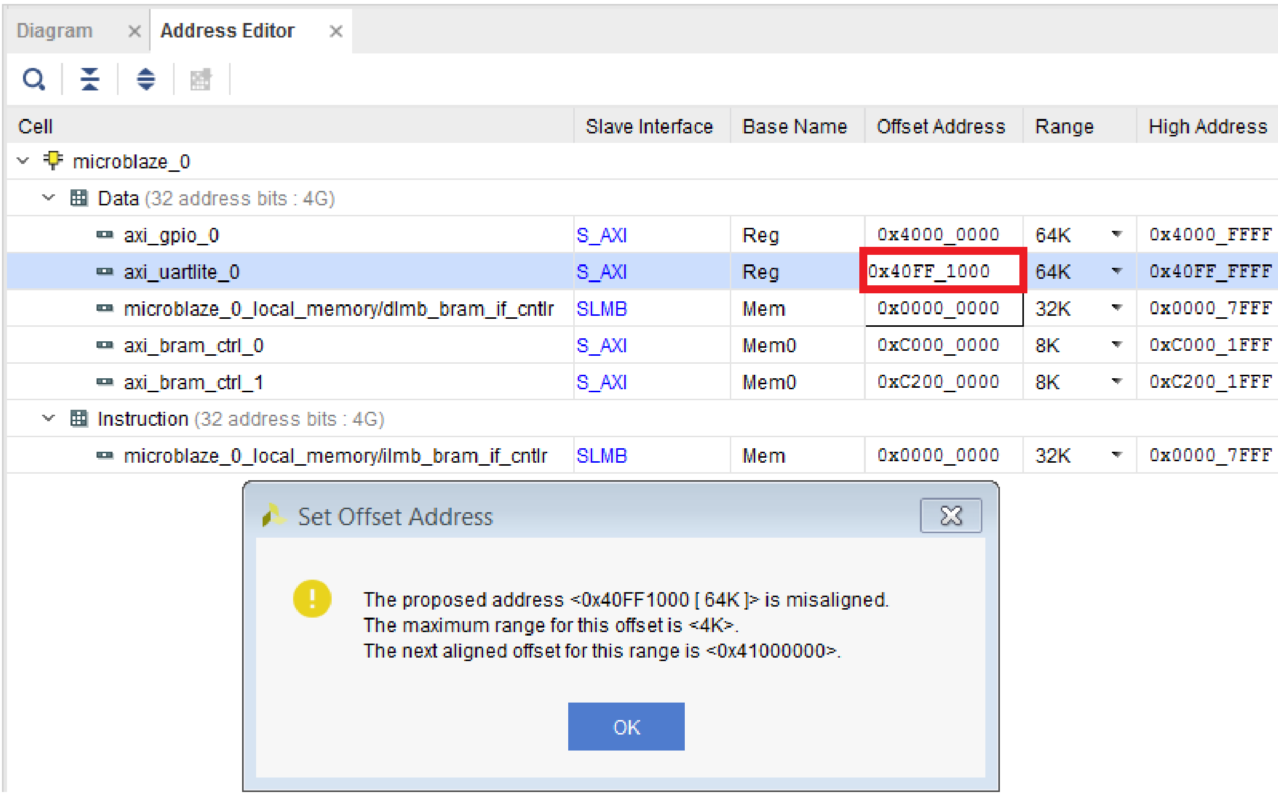
- Offset Address: Describes the offset from the start of the address block.

As an example the addressable range for data and instruction address spaces are 4G each in the following figure. The address space starts at `0x00000000` and ends at `0xFFFFFFFF`. Within this address space the `axi_uartlite_0` can be addressed to a range starting at offset `0x40600000`, `axi_gpio_0` can be addressed starting at offset `0x40000000` and so forth. This field is automatically populated as the slaves are mapped in the address space of the master; however, they can be user-specified.

The Offset Address and the Range fields are interdependent. The Offset Address field must be aligned with the Range field. Alignment implies that for a range of 2^N the least significant bits of the starting offset must have at least N 0's.

As an example, if the range of a slave segment happens to be 64K or 2^{16}, the Offset Address must be in the form `0xXXXX0000`. This means the lowest 16-bits need to be 0's. If this field is not set correctly, the error message, shown in the following figure, displays.

Figure 81: **Example of Misaligned Offset Address**



The figure shows an offset address is set with only 12 0's in the least significant bits. Only a range of 4K or 2^{12} can be accommodated by the proposed offset address. Therefore, a message opens informing the user that the address is misaligned. The message also provides where the next offset address can be set based on the current memory map.

- Range: Specifies the total range of the address block for a particular slave. This field is typically populated based on a parameter in the component.xml file for an IP. This can also be changed by clicking the drop-down menu and selecting the appropriate value for this field.

The Range and the Offset Address fields are interdependent, and as described in the Offset Address field previously, the 2^N Range field must be aligned with the N number of least significant bits of the Offset Address field.

To avoid a address misalignment, the addressing algorithm offers the range choice based on the Offset Address for that particular segment and assignment of other segments within the overall memory map.

Figure 82: **Range Alignment in Address Editor**

The figure shows the Range for axi_bram_ctrl_0 can only go up to 64K within the 0xC001_0000 and 0xC001_1FFF. If the user needs a bigger Range value then, the overall memory map of the design will need to be adjusted accordingly.

- High Address: Adjusts itself based on the Offset Address and the Range value. This is the last addressable address in a particular assigned segment.

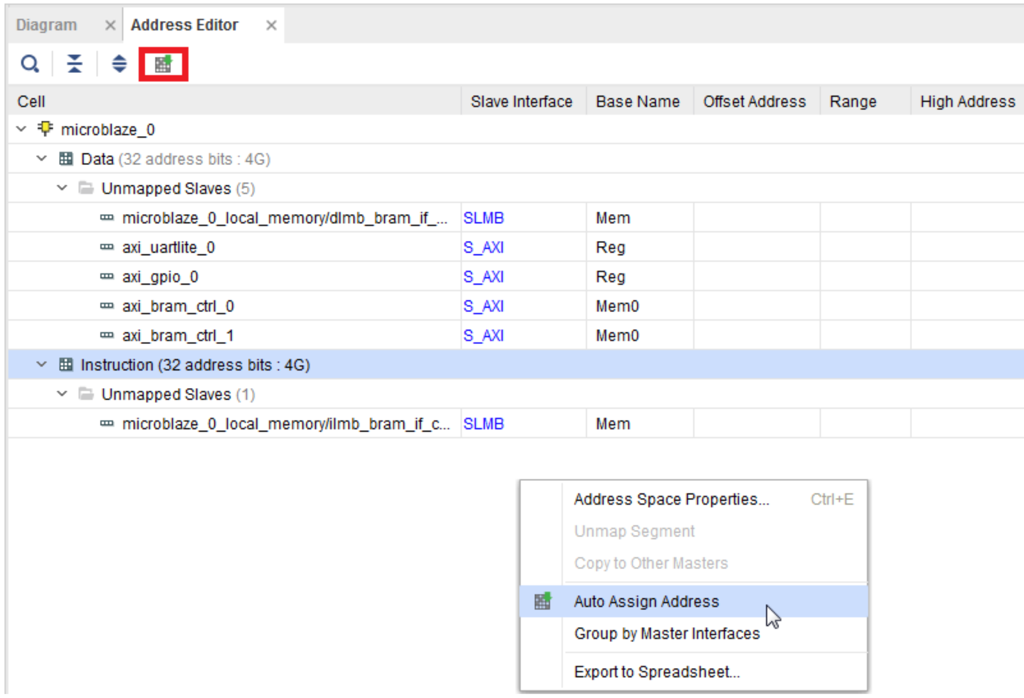
Memory Mapping Using the Address Editor

While memory block assignments happens automatically as the slave interfaces are connected to master interfaces in the BD, the mapping can also be done manually in the Address Editor.

Auto-Assigning Addresses

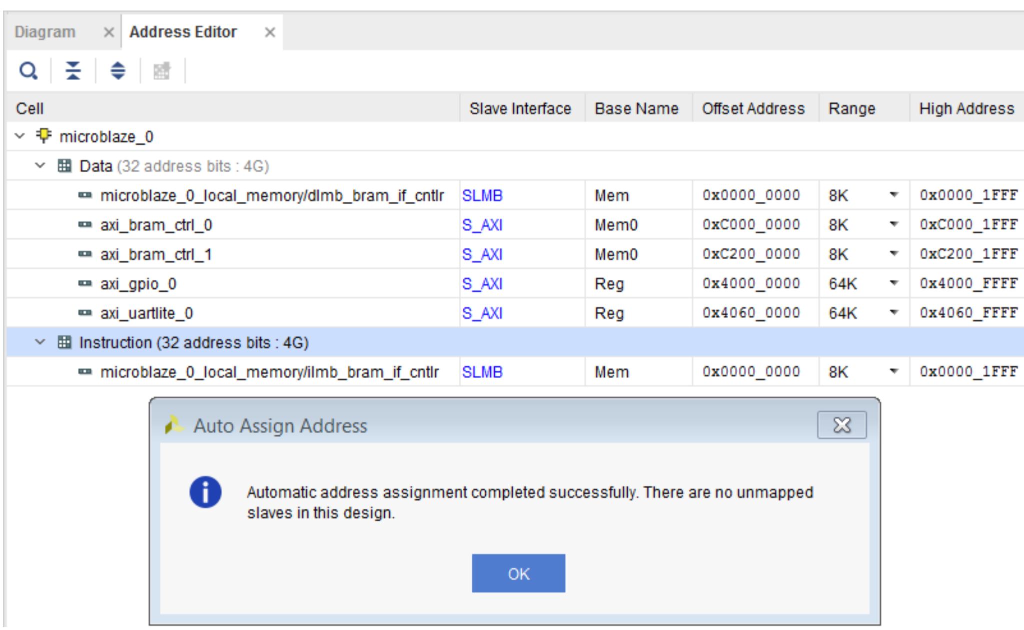
To map all the slave segments at once, right-click anywhere in the Address Editor and select **Auto Assign Address** or click the **Auto Assign Address** button on the BD toolbar as shown in the following figure.

Figure 83: **Auto Assign Address Command**



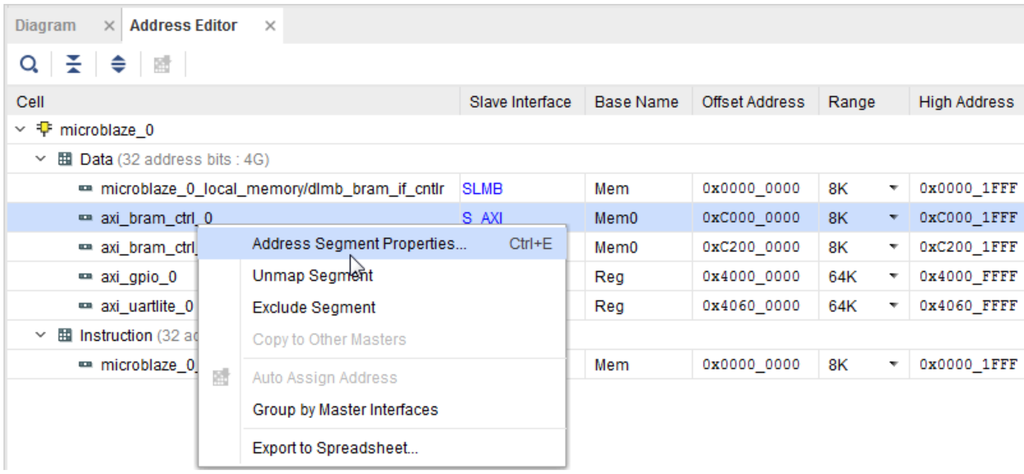
This maps the slave segments as shown.

Figure 84: **memory map After Address Block Auto Assignment**



After the slave segments are mapped, several options are presented to for other actions using a right-click on a mapped address segment, as shown in the following figure.

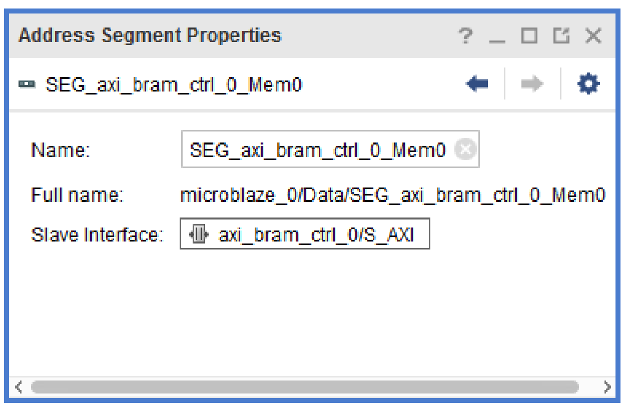
Figure 85: **Address Editor Options**



Address Segment Properties

The Address Segment Properties shows the details of the address segment in the Address Segment Properties window, shown in the following figure.

Figure 86: **Address Segment Properties Window**



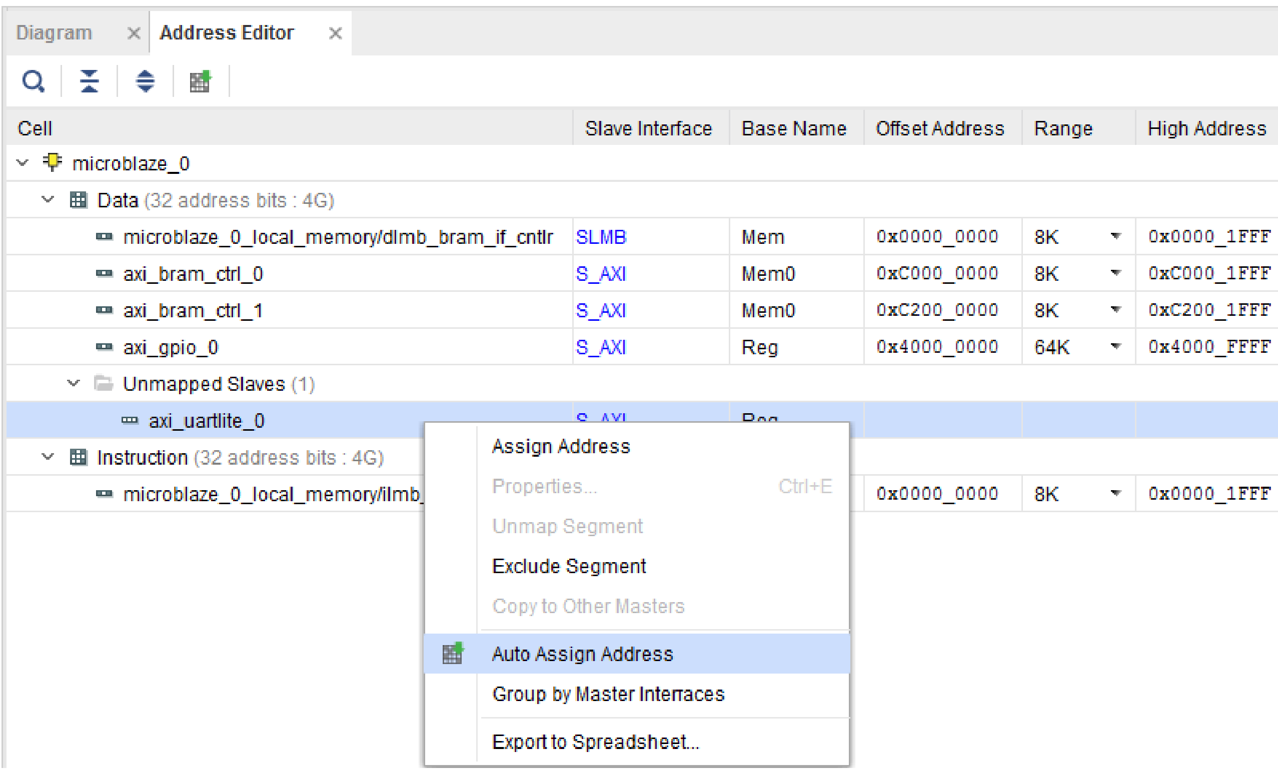
The fields that this window shows are as follows:

- Name: Shows the name of the master segment that was automatically assigned. This name can be user-specified.
- Full name: Is not editable, and shows the full name of the mapped slave segment.
- Slave Interface: Shows the slave interface of the peripheral that references the slave segment.

Unmap Segment

A mapped address segment can be unmapped by selecting Unmap Segment from the context menu. This address segment then shows up in the Unmapped Slaves folder, as shown in the following figure. You can also right-click, and select Assign Address (which maps only the selected address) or Auto Assign Address (which assigns all unmapped address segments in the design).

Figure 87: **Auto Assigning Unmapped Segments**



Exclude Segment

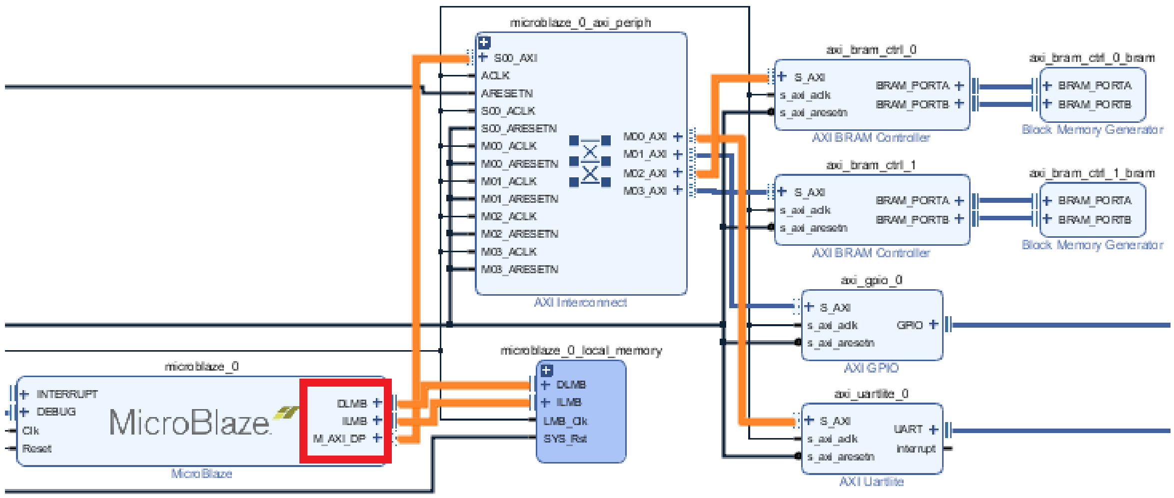
Excluding a segment makes a mapped segment unaddressable to the master in question. This is typically done when multiple masters are present in the design and the user wants to control which masters should access which slaves. See Sparse Connectivity for more information.

Group by Master Interfaces

Selecting the **Group by Master Interfaces** groups the master segments within an address space by the master interfaces through which they are accessed by the master.

For example, the MicroBlaze processor in the following BD, (shown in the following figure), has three different master interfaces accessing various address segments: `DLMB`, `ILMB`, and `M_AXI_DP` within the Data Address Space.

Figure 88: **MicroBlaze Block Grouped by Master Interfaces**



Selecting the Group by Master Interfaces from the context menu, re-arranges the different address segments in the table under the master interfaces tree.

Figure 89: **Address Editor with Grouped Address Segments under Master Interfaces**

Diagram × | Address Editor ×

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
DLMB					
microblaze_0_local_memory/dlmb_bram_if_...	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
M_AXI_DP					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
axi_bram_ctrl_1	S_AXI	Mem0	0xC200_0000	8K	0xC200_1FFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
Instruction (32 address bits : 4G)					
ILMB					
microblaze_0_local_memory/ilmb_bram_if_c...	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF

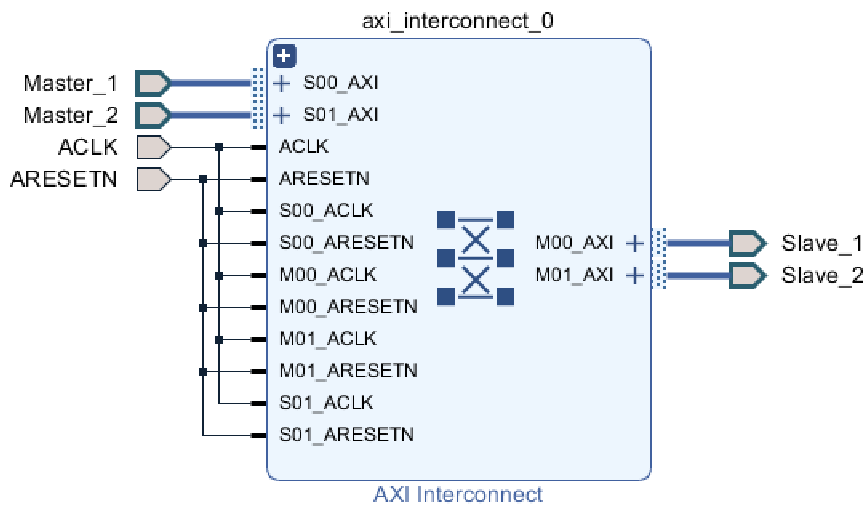
Sparse Connectivity

In a multiple master design users might want to specify slaves that could potentially be accessed by all masters or by certain masters only. This feature of memory mapping in IP integrator is called sparse connectivity.

Excluding Address Segment from a Memory-Mapped Master

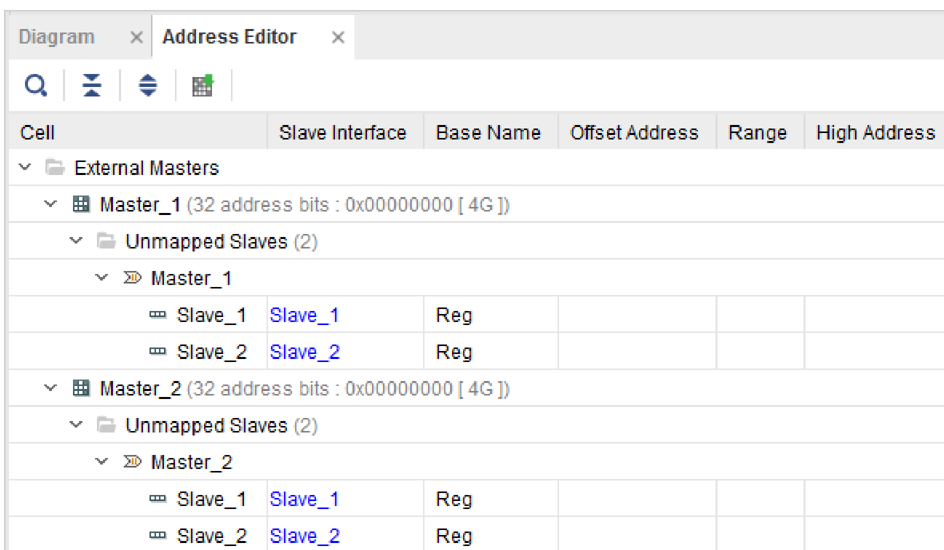
The following figure is a BD with two masters.

Figure 90: **Multiple Master and Slave Example**



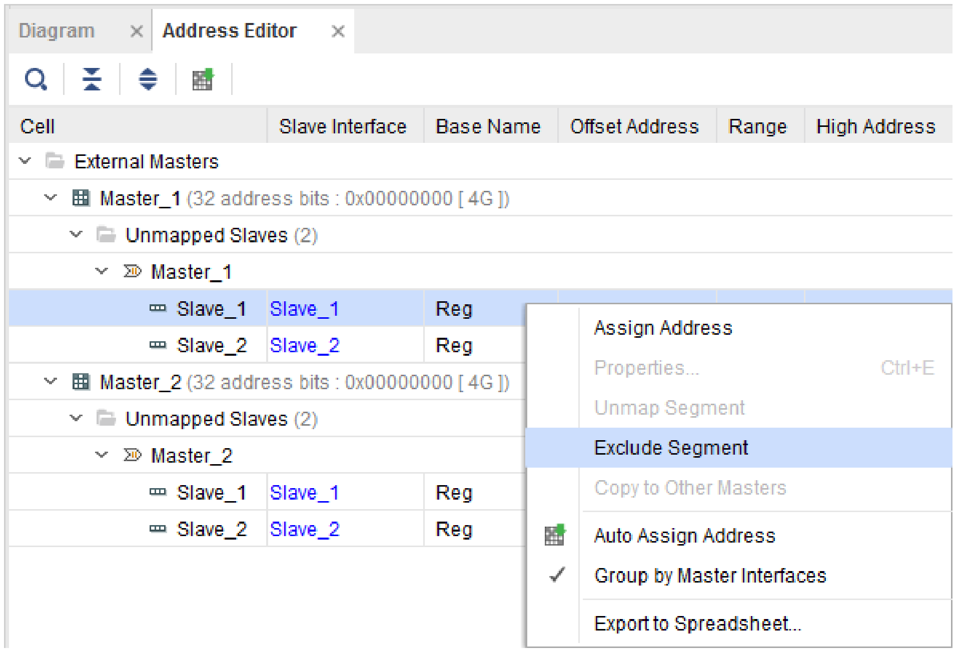
In the following figure, there are two masters, `Master_1` and `Master_2`, accessing two slaves, `Slave_1` and `Slave_2` using the same interconnect.

Figure 91: **Address Editor with Multiple Master Memory Map**



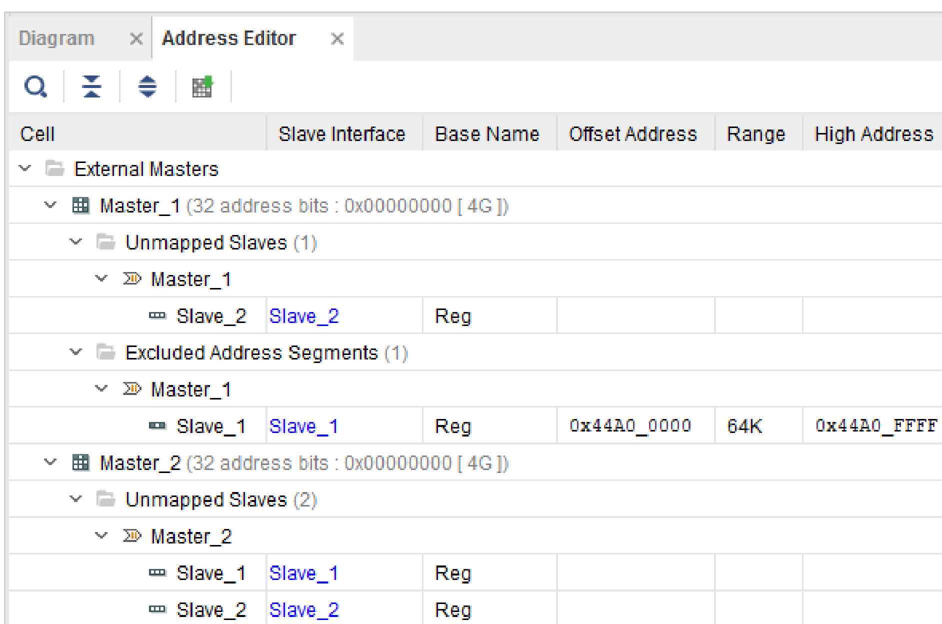
For an example, assume that `Master_1` must access `Slave_2` only, and `Master_2` needs to access both `Slave_1` and `Slave_2`. To exclude `Slave_1` from the memory map of `Master_1`, right-click `M00_AXI` and select Exclude Segment, as shown in the following figure.

Figure 92: **Exclude Segment Command**



This action excludes the segment by showing the segment under the `Excluded Address Segments` folder as shown in the following figure.

Figure 93: **Excluded Address Segment in Address Editor**



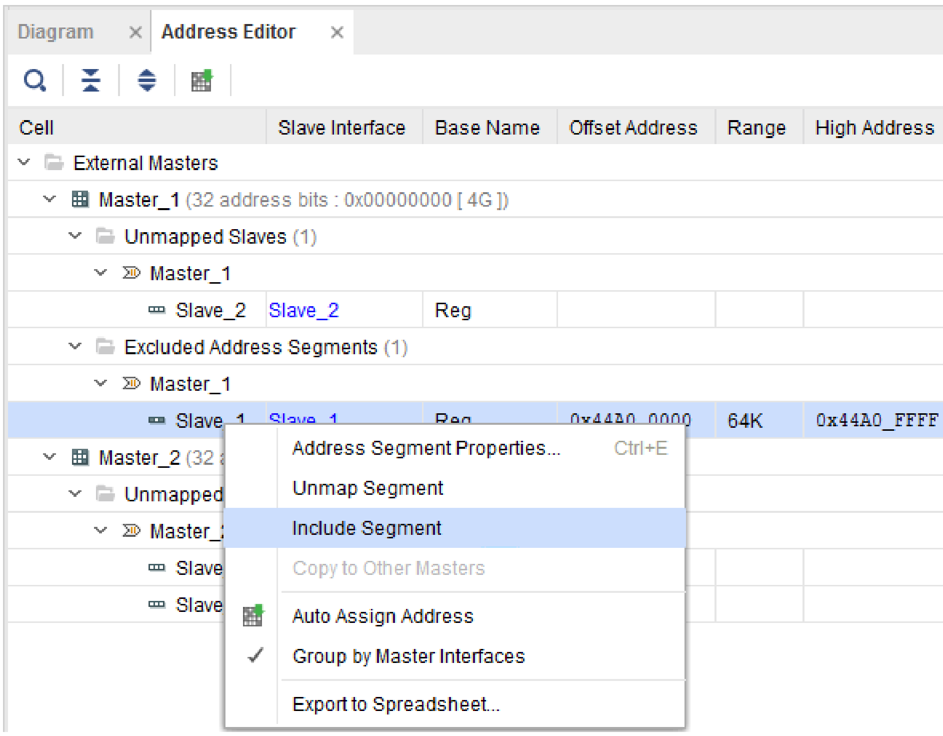
You can exclude both mapped and unmapped slaves.

IMPORTANT! *An excluded master segment still occupies a range within the address space despite it being inaccessible by the master.*

If, after excluding a slave within a master address space, one attempts to manually place another slave to address that overlaps with the excluded slave, an error occurs during validation.

An excluded segment can be added back to the Master by selecting Include Segment from the context menu as shown in the following figure.

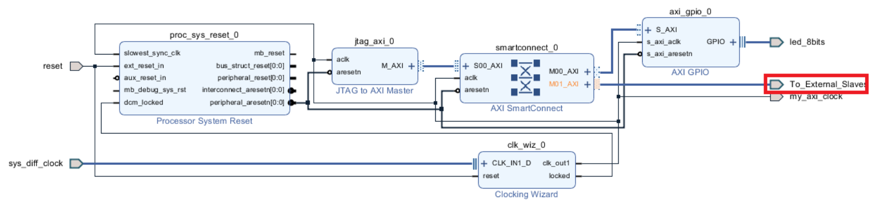
Figure 94: **Including an Excluded Segment into Master memory map**



Assigning Multiple Address Ranges for External Segments

Multiple address ranges can be assigned to external Master Port that can connect to several Slaves outside of the IP integrator design environment. Consider the following example where a master can connect to several external slaves.

Figure 95: **Example of AXI Master Accessing Multiple Slaves Outside IP Integrator**



It is represented as a virtual slave segment, M_AXI/Reg, as shown in the following figure, in the address editor. This segment can be mapped into the address spaces of masters in the diagram, in this case `jtag_axi_0`.

Figure 96: **Address Editor View of AXI Master Accessing Multiple Slaves Outside IP Integrator**

Diagram | Address Editor

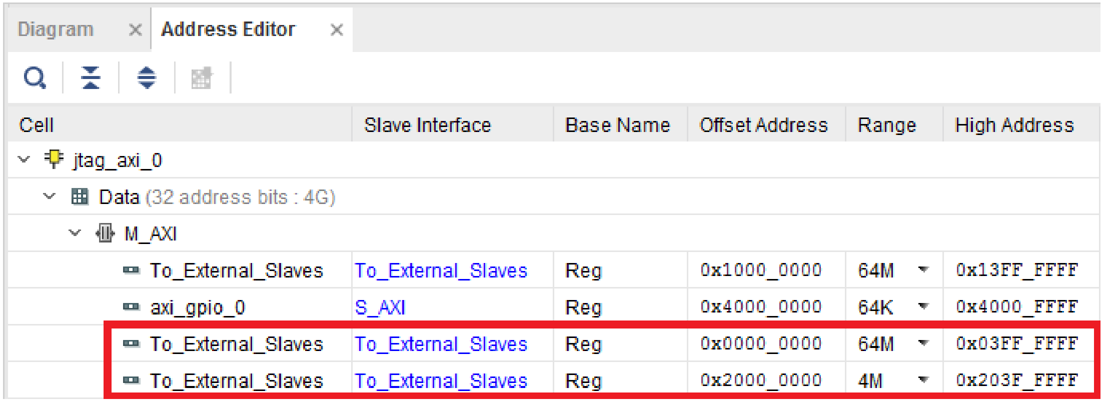
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
jtag_axi_0					
Data (32 address bits : 4G)					
M_AXI					
To_External_Slaves	To_External_Slaves	Reg	0x1000_0000	64M	0x13FF_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF

The offset of this segment is the offset that the master `jtag_axi_0` uses to initiate transactions to slaves that are connected to the `To_External_Slaves` interface. This interface can also be used to access other slaves that are not necessarily within the same offset and range by creating other segments as described in the following assign_bd_address Tcl command:

```
assign_bd_address -external -dict {offset 0x00000000 range 64M offset
0x20000000
range 4M} [get_bd_addr_segs /jtag_axi_0/Data/SEG_M_AXI_Reg] -
target_address_space
[get_bd_addr_space /jtag_axi_0]
```

Executing this Tcl command creates two separate address spaces, one at `0x00000000` with a range of `64M`, and the second one at `0x20000000` with a range of `4M`. Another way to look at this feature is to assume that the `jtag_to_axi master` in this case, needs to address other slaves through the same slave interface `To_External_Slaves`.

Figure 97: **Address Editor View of AXI Master Accessing Multiple Slaves Outside IP Integrator**



Common Addressing-Related Critical Warnings and Errors

Common addressing-related Critical Warnings, and Errors are, as follows:

```
[BD 41-971] “Segment <name of segment> mapped into <address space> at
Offset[Range]
overlaps with <name of segment> mapped at Offset [Range].
```

This message is typically thrown during validation. Each peripheral must be mapped into a non-overlapping range of memory within an address space.

```
[BD 41-1356] Address block <name of slave segment> is not mapped into <name
of
address space>. Please use Address Editor to either map or exclude it.
```

This message is typically thrown during validation. If a slave is accessible to a master, it should be either mapped into the address space of the master or excluded.

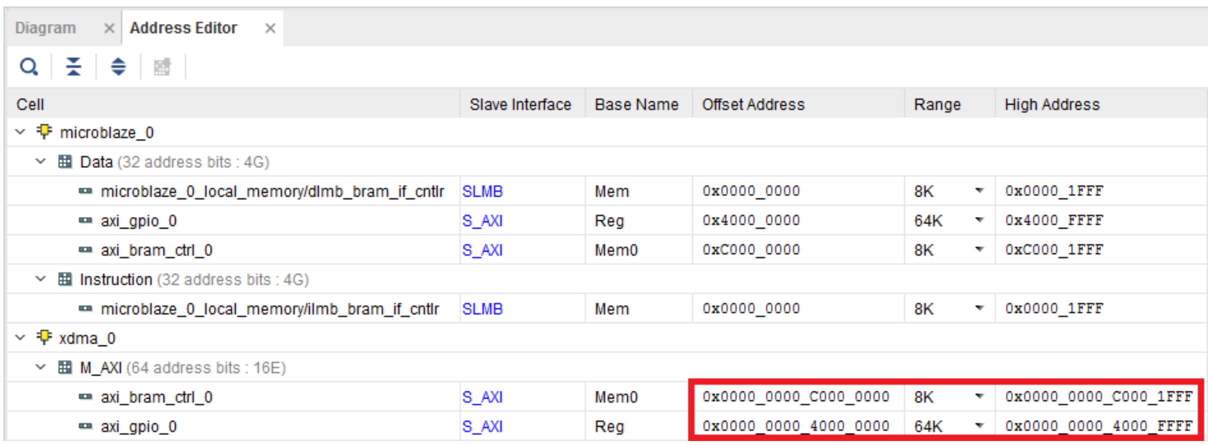
```
[BD 41-1353] <name of slave segment> is mapped at disjoint segments in
master <name
of address space> at <memory range> and in master <name of address space>
at <memory
range>. It is illegal to have the same peripheral mapped to different
addresses
within the same network. Peripherals must either be mapped to the same
offset in all
masters, or into addresses that are apertures of each other or to
contiguous
addresses that can be combined into a single address with a range that is a
power of
2.
```

This message is typically thrown during validation. Within a network defined as a set of masters accessing the same set of slaves connected through a set of interconnects, each slave must be mapped to the same address within every master address space, or apertures or subsets of the largest address range.

Support for Address Width 64-bits and Greater

Address width greater than 64-bits is supported in IP integrator as can be seen in the following figure.

Figure 98: **Support for 64-bit Address Width**



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
xdma_0					
M_AXI (64 address bits : 16E)					
axi_bram_ctrl_0	S_AXI	Mem0	0x0000_0000_C000_0000	8K	0x0000_0000_C000_1FFF
axi_gpio_0	S_AXI	Reg	0x0000_0000_4000_0000	64K	0x0000_0000_4000_FFFF

Chapter 4

Working with Block Designs

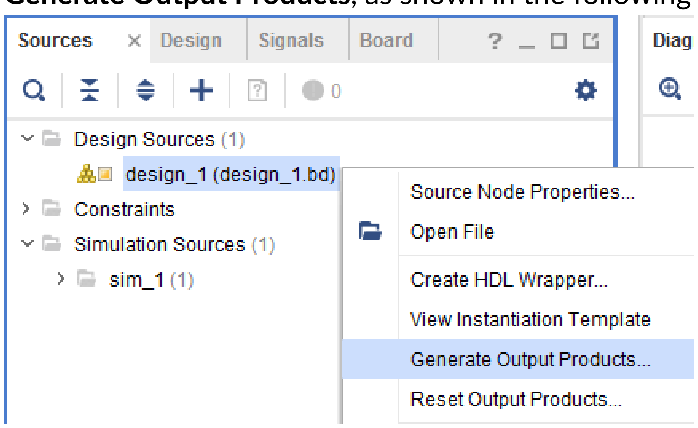
At this point, you should know how to create a block design (BD), populate it with IP, make connections, assign memory address spaces, and validate the design. This chapter describes how to work with BDs, creating the necessary output files for synthesis and simulation, adding a BD to a top-level design, and exporting the BD to the software development toolkit (SDK) for embedded processor designs.

Generating Output Products

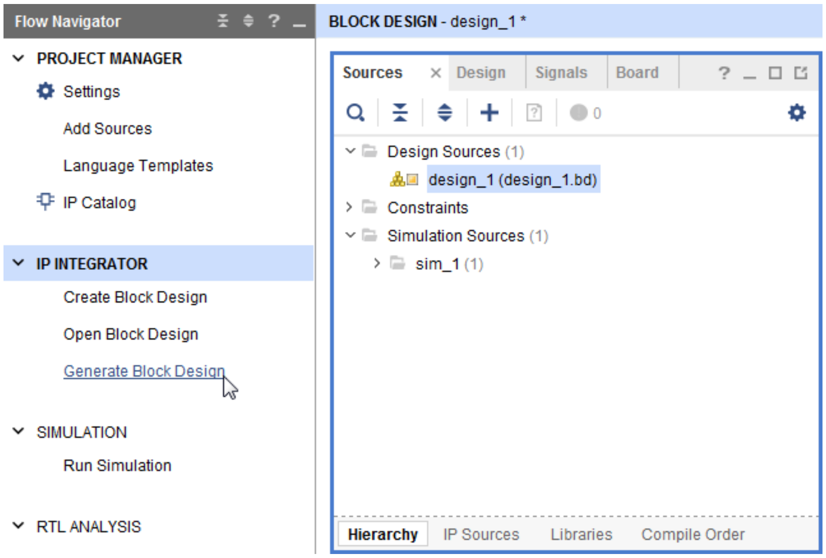
After the BD is complete and the design is validated, you must generate output products for synthesis and simulation, in order to integrate the BD into a top-level RTL design. The source files and the appropriate constraints for all the IP are generated and made available in the Vivado Integrated Design Environment (IDE) Sources window.

Output files are generated for a BD based upon the Target Language that you specified during project creation, or in the Settings dialog box. If the source files for a particular IP cannot be generated in the specified target language a message displays in the Tcl Console.

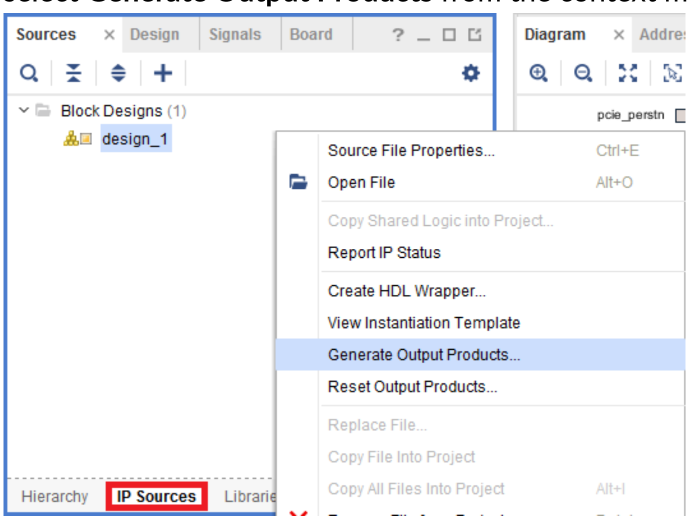
1. To generate output products, in the Vivado sources pane, right-click the BD and select **Generate Output Products**, as shown in the following figure.



Alternatively, click **Flow Navigator → IP Integrator → Generate Block Design**, as shown in the following figure.



2. To generate the output product from the IP Sources tab, select and right-click the BD, and select **Generate Output Products** from the context menu as shown in the following figure.



Generating the output products generates the top-level netlist of the BD. The netlist is generated in the HDL language specified by the **Settings → General → Target Language** for the project.

Using the Generate Output Products Dialog Box

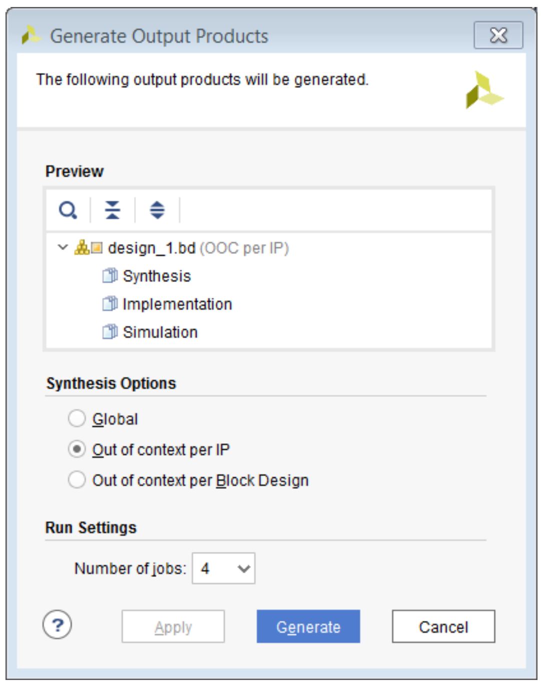
The Vivado IDE generates output products for three different modes:

- Global: Used for generating output products used in top down synthesis of the whole design. This is essentially disable out-of-context synthesis for the BD, and simply synthesizes it with the whole design.

- Out of context per IP: Generates the output product for each individual IP used in the BD, and a DCP is created for every IP used in the BD. This option can significantly reduce synthesis run times because the IP cache can be used with this option to prevent Vivado synthesis from regenerating output products for specific IP if they do not change. For more information on using the IP Cache, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896).
- Out of context per Block Design: This lets you synthesize the complete BD separately from, or *out of the context* of, the top-level design by generating a design checkpoint for the BD itself. This option is generally selected when third-party synthesis is used.

The following figure shows the Generate Output Products dialog box for a BD.

Figure 99: **Generate Output Products Dialog Box**

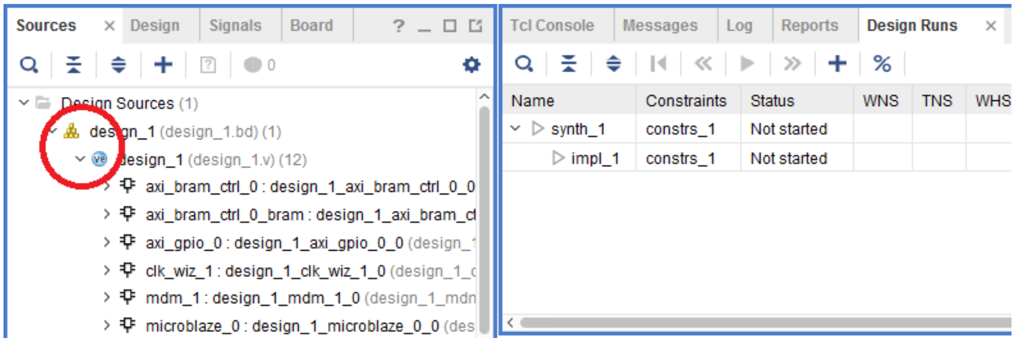


TIP: *The default mode of Synthesis is out-of-context (OOC) per IP, and IP caching is also enabled by default. This combination reduces synthesis demands.*

Global Synthesis

When this mode is chosen, a synthesized design checkpoint (DCP) is created for the entire top-level design, but not for the BD or for individual IP used in the BD. The entire BD is generated in the top-down synthesis mode. You can see this in the Design Runs window, where only one synthesis run is defined.

Figure 100: **Design Runs window for Global Synthesis**



The Tcl commands used to generate output products with the Global Synthesis mode are as follows:

```
set_property synth_checkpoint_mode None [get_files <name_of_bd>.bd]
generate_target all [get_files <name_of_bd>.bd]
```

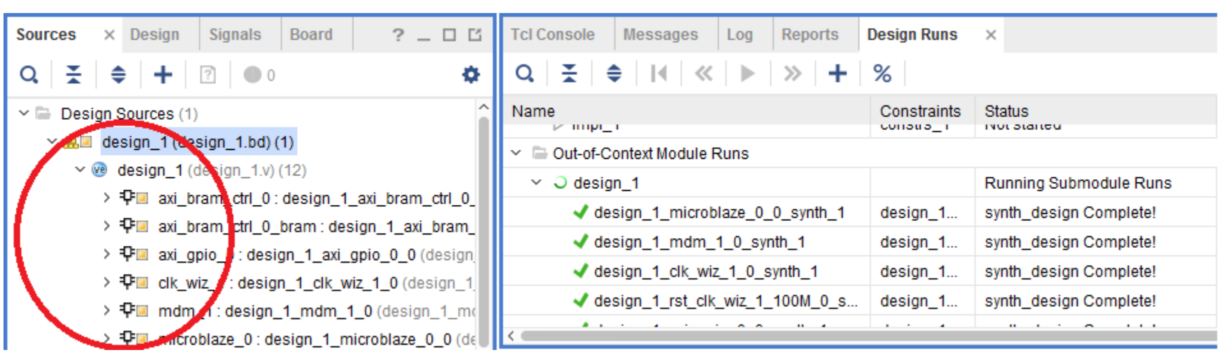
Out-of-Context per IP

This mode creates an out-of-context (OOC) synthesis run and DCP for every IP that is instantiated in the design. Notice that each IP in the BD is also marked with a filled square that indicates the IP is marked as OOC.

The Design Runs window lists synthesis runs for each IP used in the BD, as shown in the following figure.

TIP: *The Design Runs window also groups the nested synthesis runs for IP used in the child block designs of Hierarchical IP as discussed in* Hierarchical IP in IP Integrator.

Figure 101: **Design Runs Window for Out-of-Context per IP Synthesis**



Generation of the individual output products in OOC per IP mode takes longer than a single global synthesis run; however, run time improvements are realized in subsequent runs because only the updated blocks or IP are re-synthesized instead of the whole top-level design. In addition, with the IP Cache enabled, Vivado synthesis can provide even greater run time improvements because the only IP to re-synthesize have been re-customized or were impacted from parameter propagation.

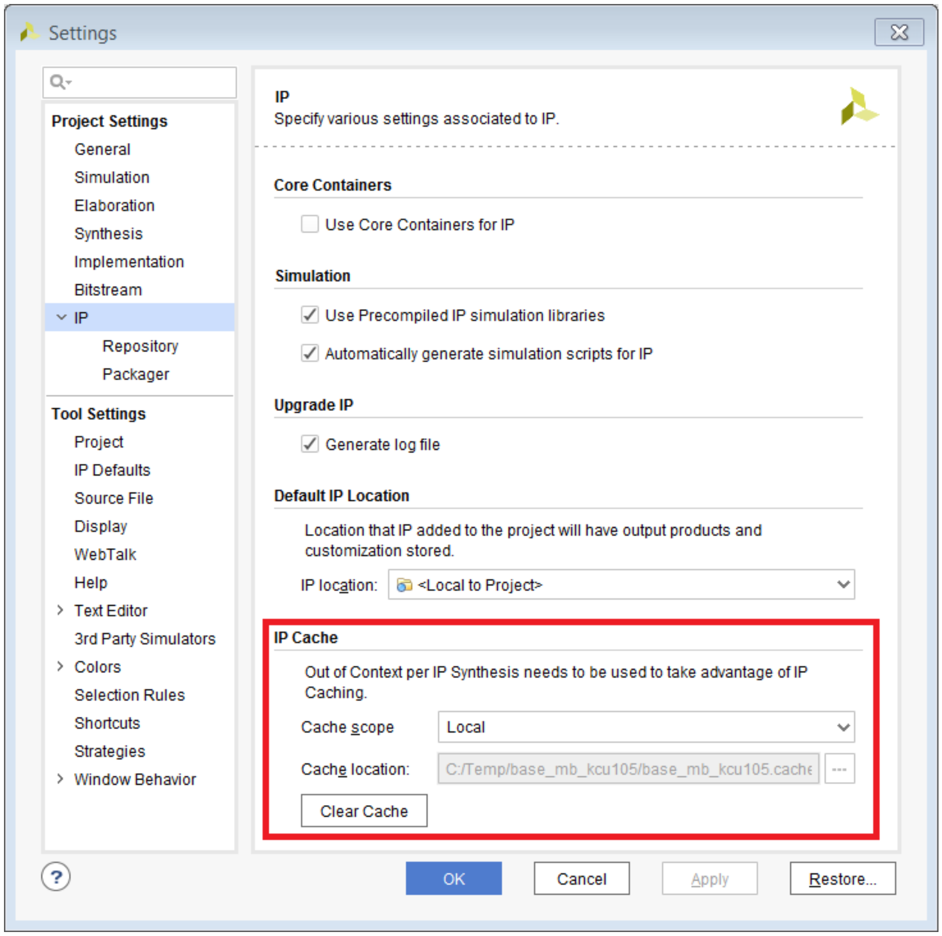
The Tcl commands used to generate output products with the Out-of-Context per IP mode are as follows:

```
set_property synth_checkpoint_mode Hierarchical [get_files <name_of_bd>.bd]
generate_target all [get_files <name_of_bd>.bd]
```

***Note*:** Concat, Slice and Constant IP blocks are always synthesized in the Global Synthesis mode. Accordingly, the Design Runs tab will not show a run for all these instances under the Out-of-Context Module runs tree.

You can enable or disable, and change the IP cache settings from the Settings > IP dialog box as shown in the following figure.

Figure 102: **Setting IP Cache in the Project Setting Dialog Box**



The Cache scope field is set to Local by default. This can be changed to Disabled or Remote as well, but it is strongly recommended that caching be turned on with either Local or Remote option for Out of context per IP synthesis mode.

With IP cache set to Local, the Vivado tools create a <project_name>.cache directory folder that holds the configuration data and synthesis results for the IP in the BD. With the Cache scope set to Remote, the IP cache folder(s) are created in the specified Cache Location.

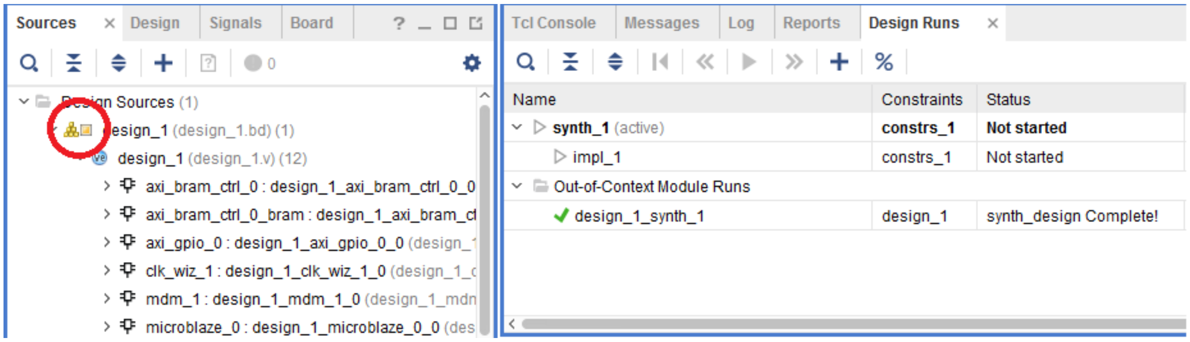
Cache data can be cleared by clicking the Clear Cache button.

Out-of-Context per Block Design

Typically used with third-party synthesis tools, this option synthesizes the BD as an OOC module, and creates a design checkpoint for the entire BD. As can be seen from the figure below, the Sources window shows that a Design Checkpoint file (DCP) was created for the BD.

Notice that the BD is also marked with a filled square that indicates the BD is marked as OOC. The Design Runs window shows the OOC synthesis run for the BD.

Figure 103: **Design Runs Window for Out-of-Context per Block Design Synthesis**



If the BD is added as a synthesized netlist to other projects through the Add Sources wizard, the DCP file is added to the project. See this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) for more information on adding BDs as design sources.

The Tcl commands used to generate output products with the Out-of-Context per Block Design mode are as follows:

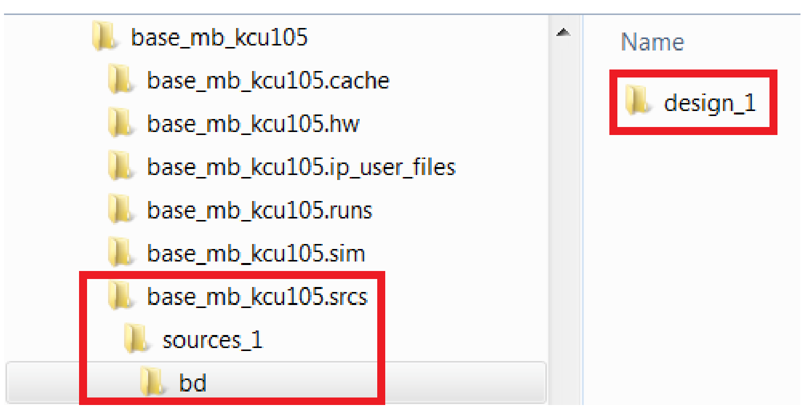
```
set_property synth_checkpoint_mode Singular [get_files <name_of_bd>.bd]
generate_target all [get_files <name_of_bd>.bd]
```

Examining Generated Output Products

The generated output products for a BD can be found in the `<project_name>/<project_name>.srcs/sources_1/bd` folder.

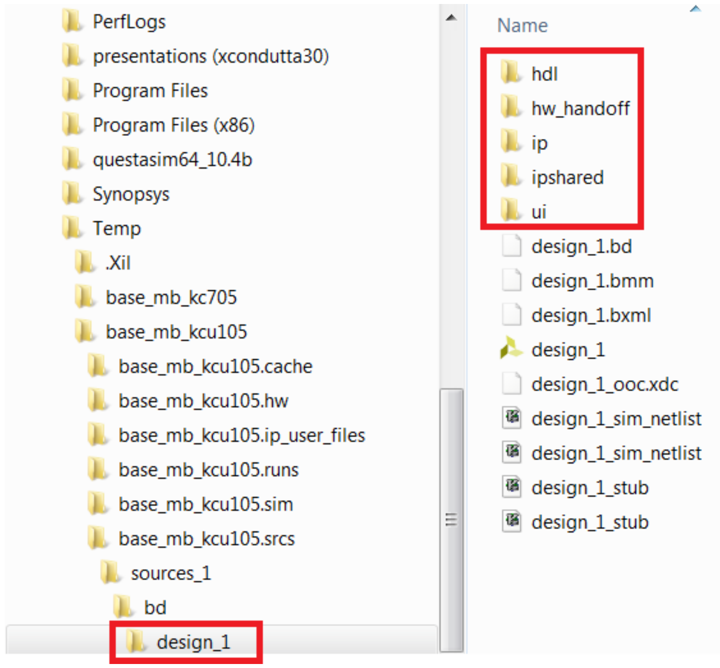
Inside the folder is a separate directory for each BD. In the following figure, `design_1` is the only BD.

Figure 104: **Locating Output Products for Block Designs**



Under the `<block_design_name>` folder, several sub-folders are located as shown in the following figure.

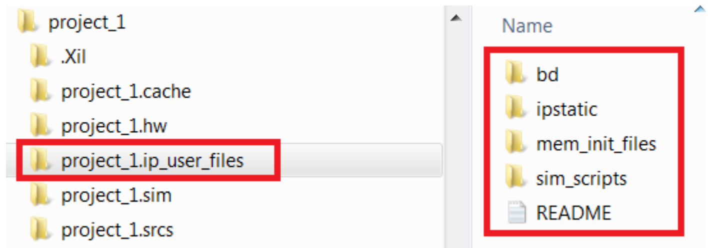
Figure 105: **Sub-Folders of a Block Design**



- `hdl`: Contains the top level netlist of the BD as well as the Vivado managed wrapper file for the BD.
- `hw_handoff`: Contains intermediate files needed for hardware handoff to SDK.
- `ip`: Contains several sub-folders, one per IP inside the BD. These IP folders may contain several sub-folders which may vary depending on the IP. Typically all the source files and constraints files delivered for the IP can be found in these sub-directories.
- `ipshared`: Contains files that are common between various IP. IP can have several sub-cores within them. Files shared by these sub-cores can be found in the `ipshared` folder.
- `ui`: This folder contains the *.ui file which has the graphical information such as coordinates of different blocks on the canvas, comments, colors and layer information.

Additionally, when the Vivado IDE generates output products for the BD it also creates a folder called `<project_name>/<project_name>.ip_user_files`, as shown in the following figure. Inside of the `<project_name>.ip_user_files` folder there are a number of folders depending on the contents of your project (IP, BDs, and so forth).

Figure 106: **Sub-Folders Under the ip_user_files Folder**



The following is a brief description of the directories that could be present in the `<project_name>.ip_user_files` folder:

- `bd`: Contains a sub-folder for each IP integrator BD in the project. These sub-folders will have support files for the various IP used in the BDs.
- `ipstatic`: Contains common IP static files from all IP/BDs in the project.
- `mem_init_files`: Is present if any IP deliver data files.
- `sim_scripts`: By default, scripts for all supported simulators for the OS selected are created for each IP and for each BD present.

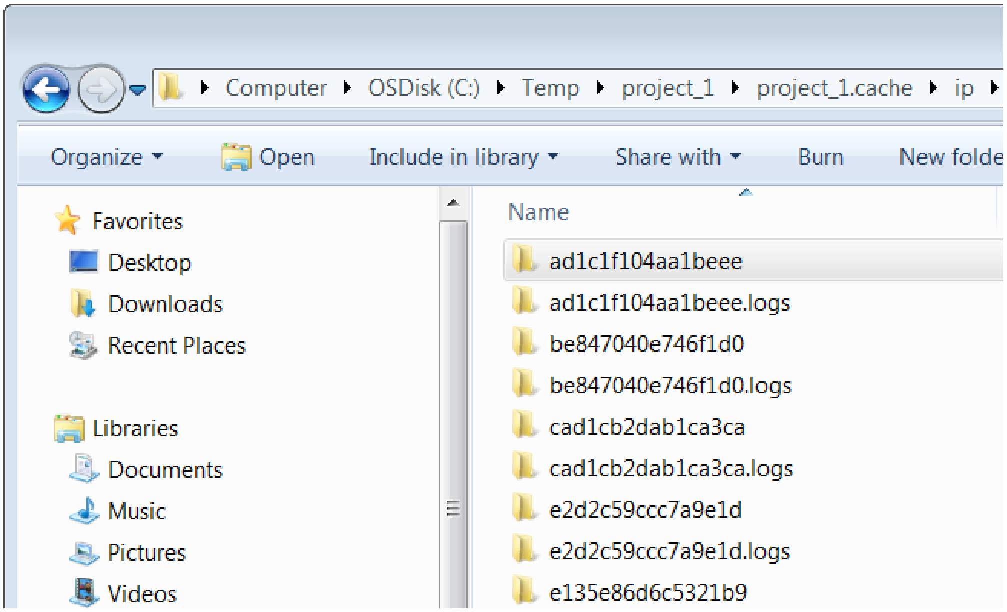
To manually export IP or BD files to the `ip_user_files` directory, you can use the export_ip_user_files command at the Tcl Console. Whenever you reset and generate an IP or BD, this command runs automatically. For more information, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

When the Output Products for a BD are generated, several status messages are flagged on the Tcl Console as shown below.

```
catch { config_ip_cache -export [get_ips -all design_1_microblaze_0_0] }

INFO: [IP_Flow 19-4993] Using cached IP synthesis design for IP
design_1_microblaze_0_0, cache-ID = ad1c1f104aa1beee; cache size = 8.220 MB.
catch { config_ip_cache -export [get_ips -all design_1_dlmb_v10_0] }

INFO: [IP_Flow 19-4993] Using cached IP synthesis design for IP
design_1_dlmb_v10_0,
cache-ID = ecf144ac474f353c; cache size = 8.220 MB.
catch { config_ip_cache -export [get_ips -all
design_1_dlmb_bram_if_cntlr_0] }

INFO: [IP_Flow 19-4993] Using cached IP synthesis design for IP
design_1_dlmb_bram_if_cntlr_0, cache-ID = be847040e746f1d0; cache size =
8.220 MB.
```

The `[IP_Flow 19-4993]` message informs the user of the cache-ID associated with the cell in the BD. The individual cache-ID folders can be found in the IP Cache location.

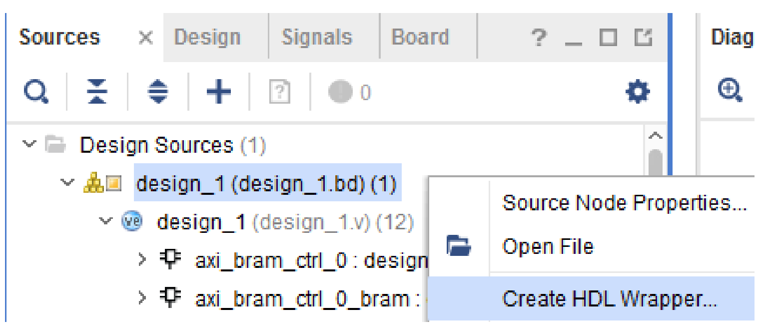
Figure 107: **Cache-ID Directories**



Integrating the Block Design into a Top-Level Design

An IP integrator BD can be integrated into a higher-level design or it can be defined as the top-level of the design hierarchy. In either case, begin by generating an HDL wrapper for the BD. Right-click the BD in the Vivado IDE Sources window and select **Create HDL Wrapper**.

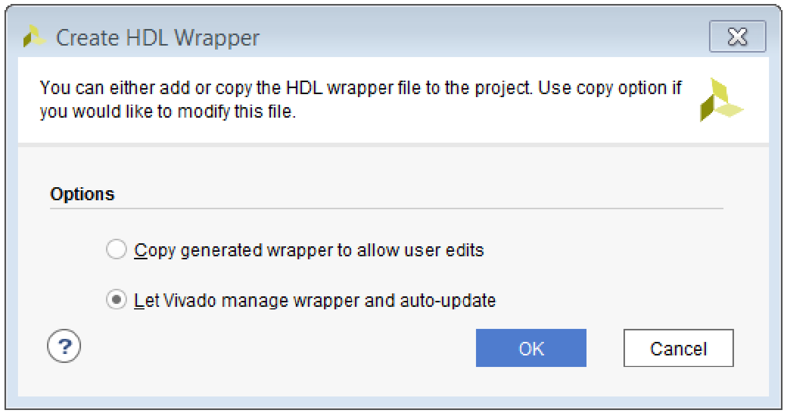
Figure 108: **Create HDL Wrapper Command**



This command generates a top-level HDL file with an instantiation template for the IP integrator BD.

The Create HDL Wrapper dialog box opens, as shown in the following figure.

Figure 109: **Create HDL Wrapper Dialog Box**



The Create HDL Wrapper options are as follows:

- Copy generated wrapper to allow user edits. When a BD is a subset of an overall design hierarchy, you must have the option to manually edit the wrapper file so you can then instantiate other design components within the wrapper file.

IMPORTANT! *You must manually update this file, or regenerate it any time the I/O interface of the block design changes.*

The copied wrapper file is written to the `<project_name>.srcs/sources_1/imports/hdl` directory.

- Let Vivado tools manage wrapper and auto-update. Use this option if the BD is the top-level of the project, or if you will not be manually editing the wrapper file.

When the Vivado tools manage the wrapper file, the file is updated every time you generate output products. The wrapper file is located in the directory `<project_name>.srcs/sources_1/bd/<bd_name>/hdl`.

Instantiating I/O Buffers

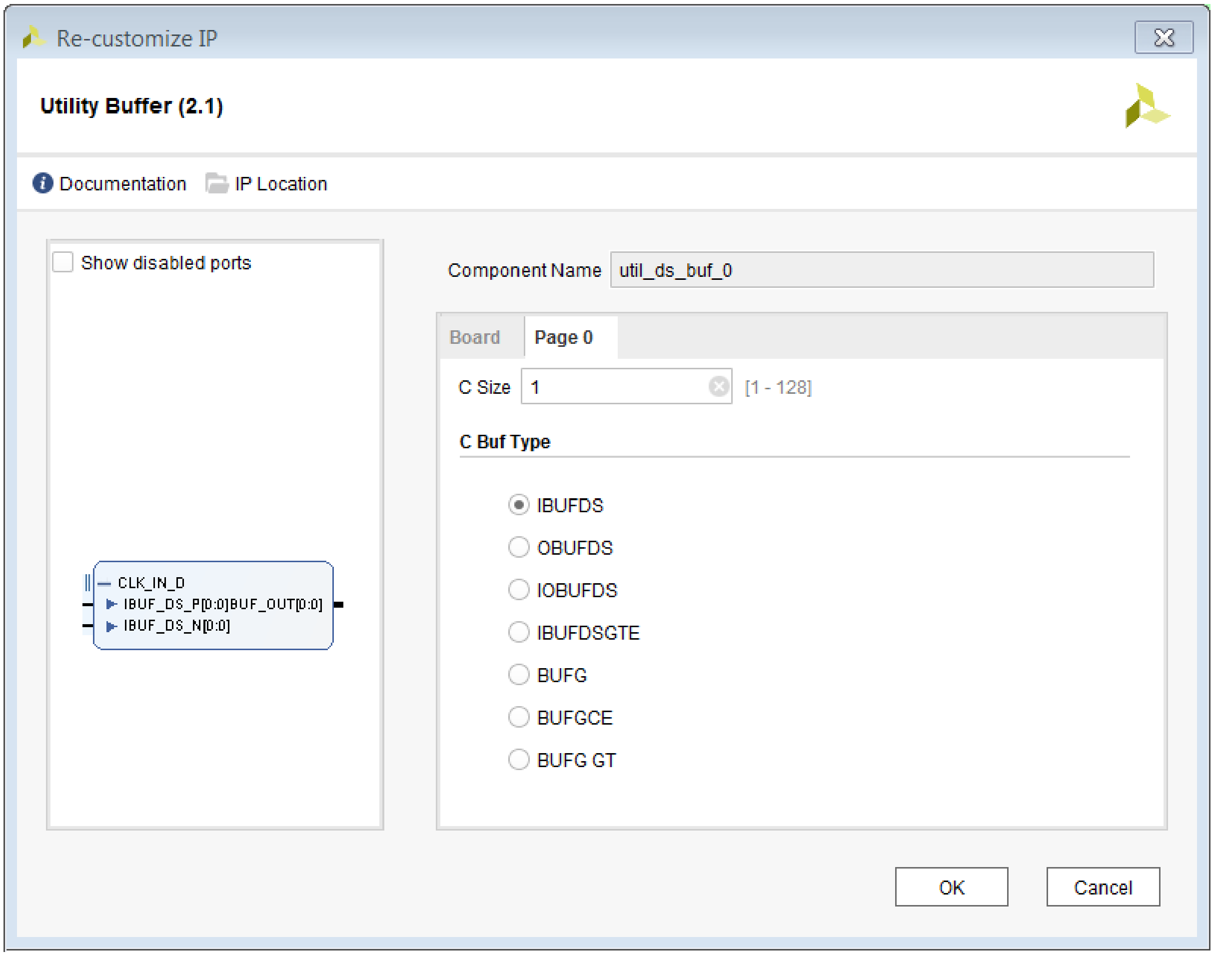
When generating the wrapper, IP integrator looks for I/O interfaces that are made external in the BD. If the tool finds external I/O, it reviews the port maps of that interface. If the tool finds three ports matching the pattern `<name>_I`, `<name>_O`, and `<name>_T`, then it instantiates an I/O buffer and connects the signals appropriately. If any of the three ports are not found, then an I/O buffer is not inserted.

Other conditions in which I/O buffers are not inserted include the following:

- If any of the `<name>_I`, `<name>_O`, and `<name>_T` ports are manually connected by the user, either by making them external or by connecting it to another IP in the design.
- If the interface has the `BUFFER_TYPE` parameter set to `NONE`.

To manually instantiate I/O buffers in the BD, you can use the Utility Buffer IP that is available in the Vivado IP catalog. This IP can be configured as different kinds of I/O buffers as shown below. See the *LogiCORE IP Utility Buffer Product Brief* (PB043) for more information.

Figure 110: **Utility Buffer IP Configuration Dialog Box**



Adding Existing Block-Designs

You can add an existing BD as a design source to a new project, either from an existing project or from a remote directory location.

Assuming that a BD was created using a project-based flow, and all the directory structure including and within the BD folder is available, the BD can be added to a new Vivado project.

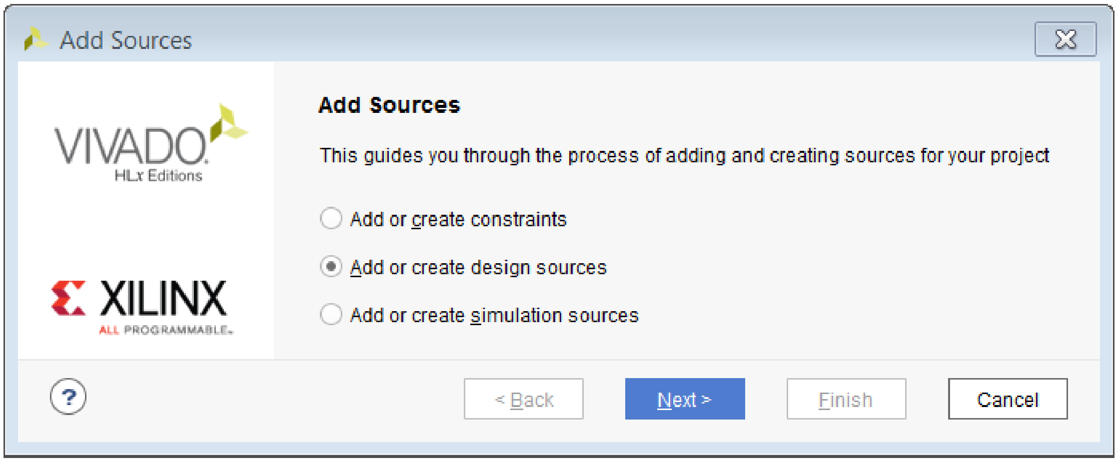
The only limitation is that the target part or platform board for the current project must be the same as the original project in which the BD was created.

IMPORTANT! *If the target devices of the projects are different, even within the same device family, the IP used in the block design will be locked, and the design must be re-generated. In that case the behavior of the new block design might not be the same as the original block design.*

1. To add a remote BD, select **Flow Navigator → Project Manager → Add Sources**.

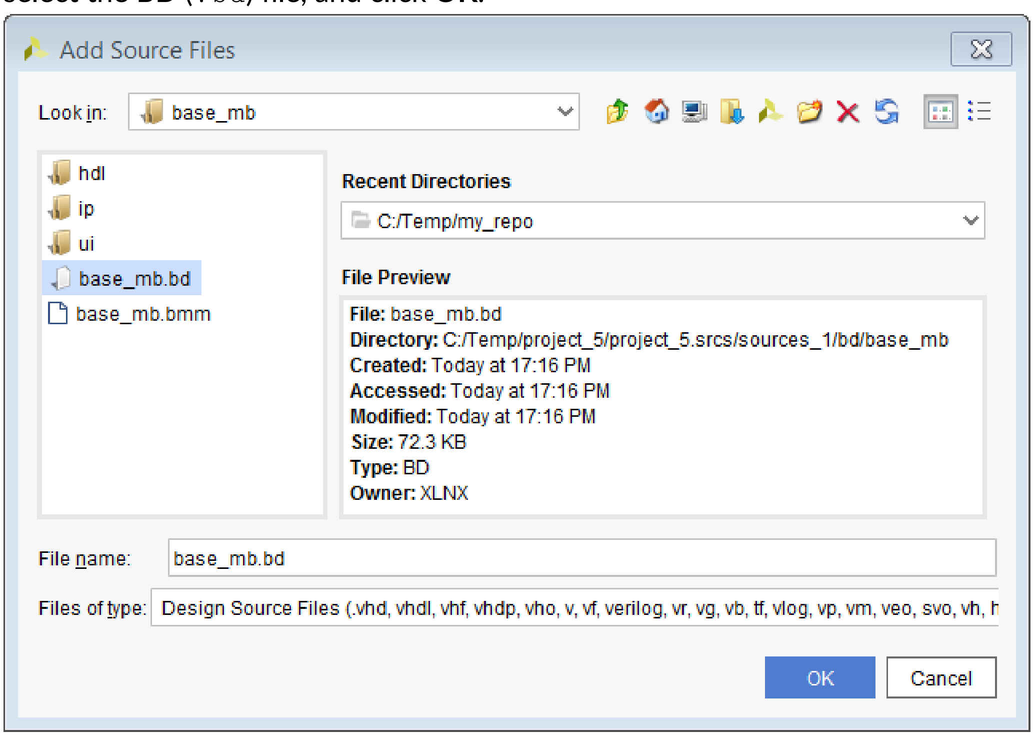
   Alternatively, you can right-click in the Sources window, and select **Add Sources**.

2. In the Add Sources wizard, select **Add Existing Block Design Sources**, as shown in the following figure, and click **Next**.

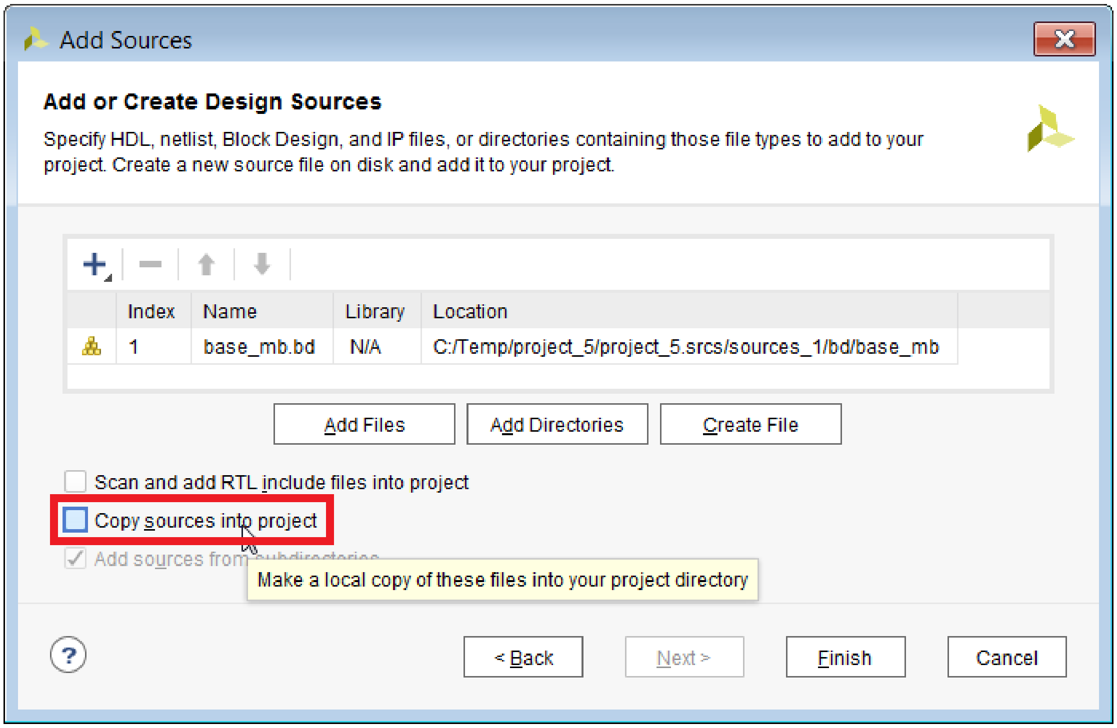


3. In the Add Existing Block Design Sources page, click **Add Files**, or click the + icon.

4. In the Add Sources File window, navigate to the folder where the block design is located, select the BD (`.bd`) file, and click **OK**.



5. In the Add Existing Block Design Sources page, you can select **Copy sources into project** as needed for your current project.



You can reference the BD from its original location, or copy it into the local project directory.

RECOMMENDED: *Managing the block design remotely is the recommended practice when working with revision control systems. See* Revision Control for Block Designs.

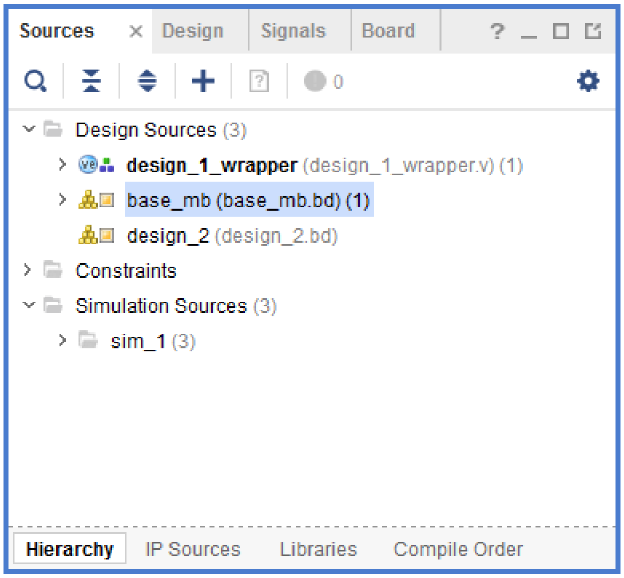
However, if someone edits the remote BD, they could inadvertently change your referenced copy. To avoid that, you can select **Copy sources into project**, as seen above, so that you can change the BD when needed, but remote users will not be able to affect your design.

You can also set the BD as read-only to prevent modification. See Adding Read-Only Block Designs for more information.

TIP: *When adding a block design from a remote location, ensure that the design is reserved for your project by copying the remote block design locally into the project.*

6. Click **Finish** to close the Add Sources wizard and add the BD to your project.

   In the Sources window, you can see the BD added under Design Sources, as shown in the following figure.



7. Double-click the BD to open it in the Vivado IP Integrator.

TIP: *You might need to update the IP used in the block design, or validate the block design, generate a wrapper, and synthesize and implement the design. These topics were previously described in this document.*

Adding Read-Only Block Designs

You can set the file permissions on existing BDs as read-only for use in other projects. This will prevent the BDs from being inadvertently modified.

If you have generated output products for the BD, you can change the file permissions on all files (using chmod 555 bd -R on Linux).

The BD, and all its output products, will be read-only. Synthesis, simulation, and implementation can be run using these files.

TIP: *On Windows you can select the files, and change file properties to read-only.*

However, if you have not generated output products for the block design (BD), you can still make the BD file read-only (using `chmod 555 bd/design_1/design_1.bd` in Linux). From this read-only you can still generate the output products needed for the design, but the BD itself cannot be changed. You can generate the output products for read-only BDs, if they have not been previously generated, provided the BD has been validated and saved.

Typically, for read-only BDs, either a user managed wrapper file or a Vivado managed wrapper file is already generated. That wrapper file should be added to the project along with the BD.

IMPORTANT! *A wrapper file cannot be generated for a read-only block design.*

Revision Control for Block Designs

Revision control systems can manage the various source files associated with Vivado IP integrator BDs, in both Project and Non-Project Mode. As BDs are developed and become more complex it is a challenge to keep track of the different iterations of the design, and to facilitate project management and collaboration in a team-design environment.

See this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) for more information on using the Vivado Design Suite with revision control software.

Exporting a Hardware Definition to SDK

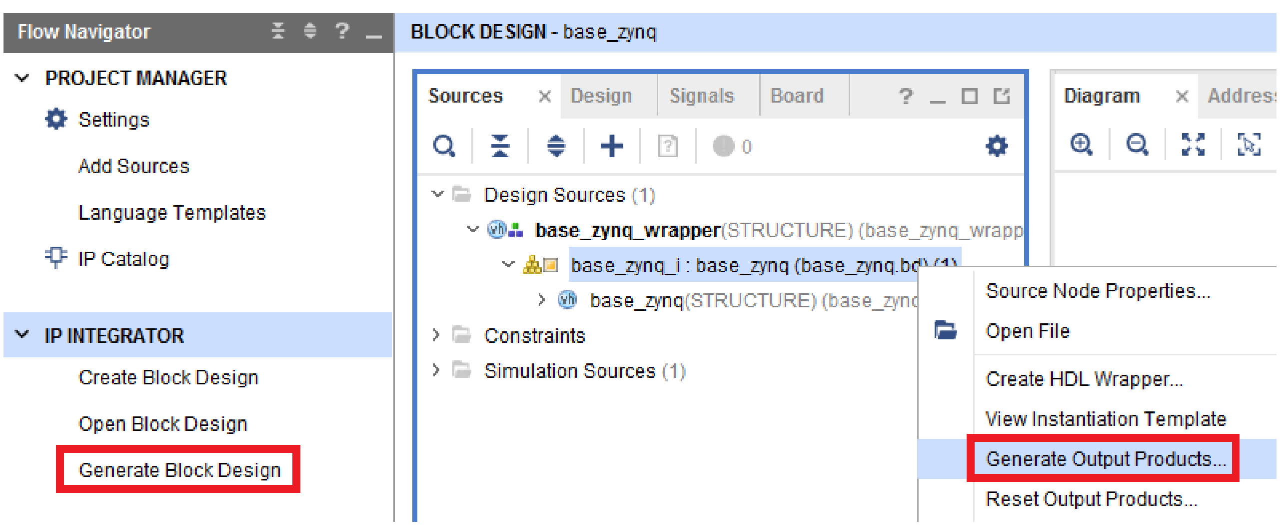
To start software development before a bitstream is created, you can export the hardware definition to the Software Development Kit (SDK) after generating the design. This exports the necessary XML files needed for SDK to interpret the IP used in the design and also exports the memory mapping from the processor perspective.

After a bitstream is generated and the design is exported, then the device can be downloaded and the software can run on the processor. The hardware can be exported at two stages in the design flow: pre-synthesis and post bitstream generation.

1. To export the hardware prior to synthesis, use the following steps:

    a. In the Flow Navigator, under IP Integrator, click **Generate Block Design**.

    Alternatively, select the BD in the Sources window, right-click and select **Generate Output Products**.



b. In the Generate Output Products dialog box, select the appropriate option, and click **Generate**.

c. To export the hardware, select **File → Export → Export Hardware**.

d. In the Export Hardware dialog box, shown in the following figure, disable **Include bitstream**, as there is no bitstream at this time.

e. Leave the Export to: field to its default value of Local to Project.

f. Click **OK**.

The following commands are executed in the Tcl console:

```
file mkdir <project_name>/<project_name>.sdk
write_hwdef -force  -file
<project_name>/<project_name>.sdk/<block_design_name>_wrapper.hdf
```

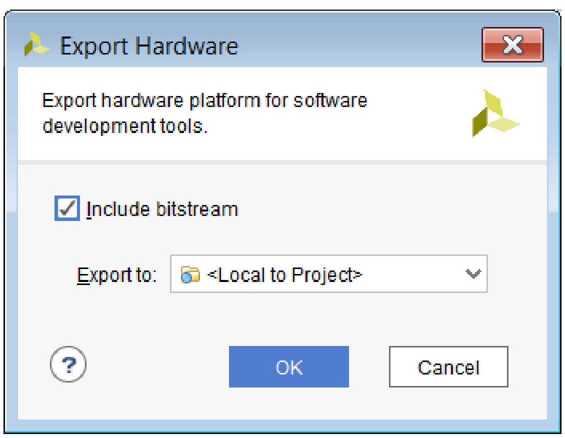
2. For exporting the hardware after bitstream generation, use the follow the steps:

a. From the menu, select **File → Export → Export Hardware**.

b. In the Export Hardware dialog box, select **Include Bitstream**.

c. Leave the Export to: field to the default value: Local to Project.

d. Click **OK**.



The following commands are executed on the Tcl Console:

```
file mkdir <project_name>/<project_name>.sdk
file copy -force
<project_name>/<project_name>.runs/impl_1/
<block_design_name>_wrapper.sysdef
```

For more information on exporting hardware, see *Generating Basic Software Platforms Reference Guide* (UG1138).

Adding and Associating an ELF File to an Embedded Design

In a microprocessor-based design such as a MicroBlaze design, an Executable and Linkable Format (ELF) file generated in the SDK (or in other software development tool) can be imported and associated with a block design in the Vivado tool. A bitstream can then be generated for the design that includes the ELF contents for use on the target hardware. There are two ways in which you can add the ELF file to an embedded object.

Adding ELF and Associating it With an Embedded Processor

To add an ELF to the project and associate it with an embedded processor, use the following steps:

1. In **Flow Navigator → Project Manager**, select **Add Sources**.

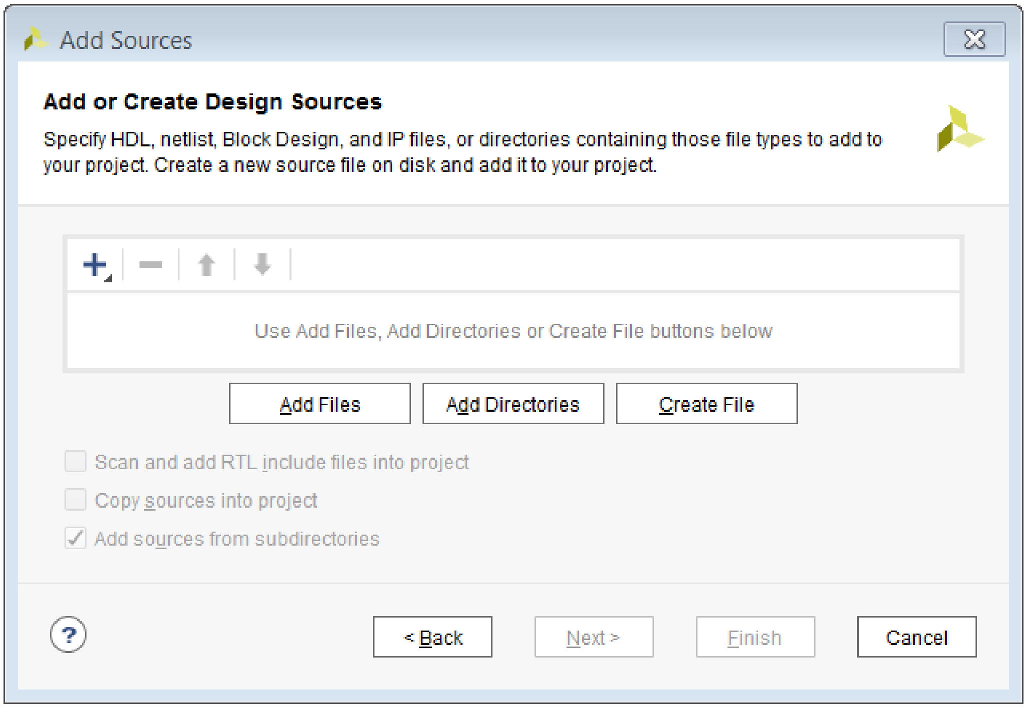
   Add or create design sources is selected by default. This option lets you add an ELF file as a design and simulation source.

TIP: *If you are adding an ELF file for simulation purposes only, select* ***Add or Create Simulation Sources****.*

---

2. Click **Next**.

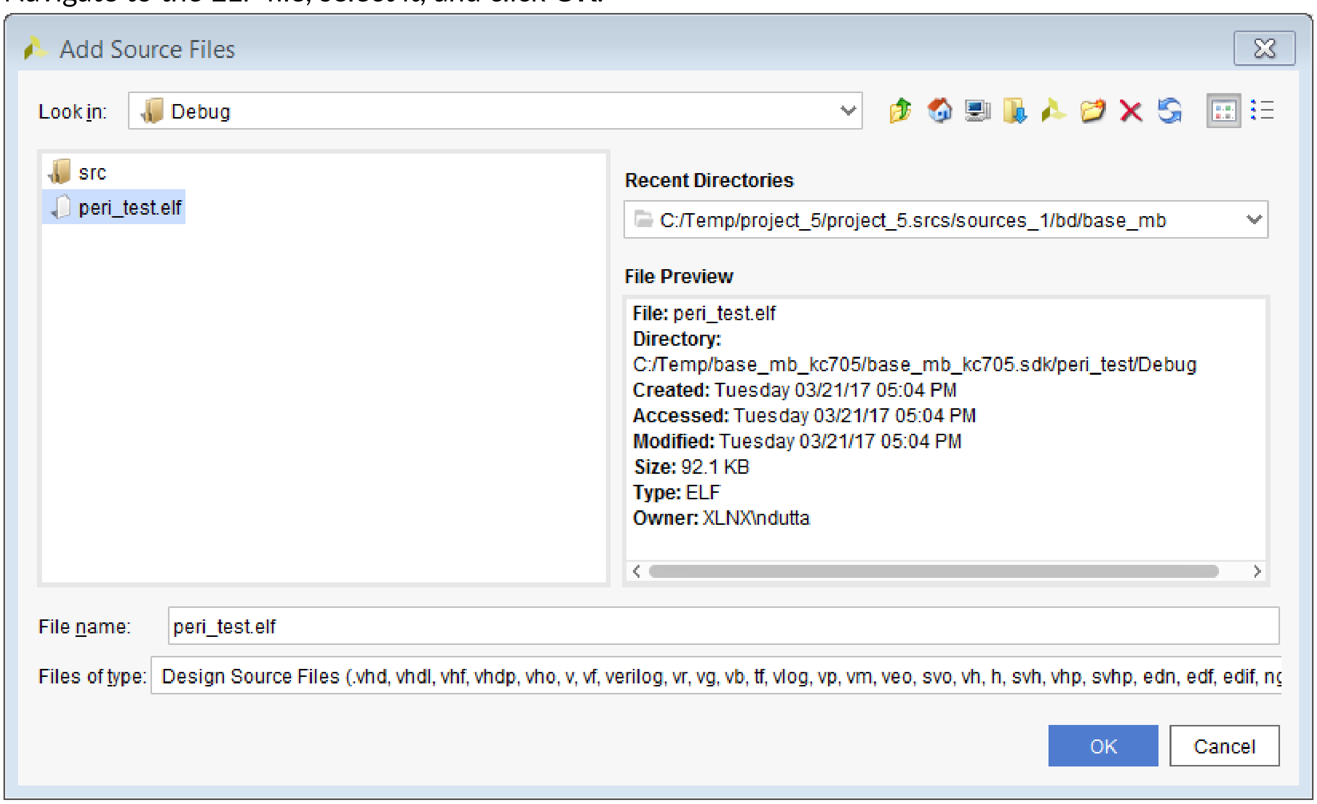
   The Add or Create Design Sources page opens as shown in the following figure.



3. Click **Add Files**.

   The Add Source Files dialog box opens, as shown in the following figure.

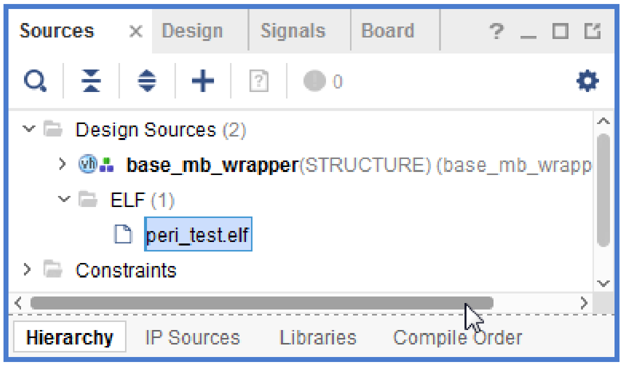
4. Navigate to the ELF file, select it, and click **OK**.



In the Add or Create Design Sources page, you see the ELF file added to the project.

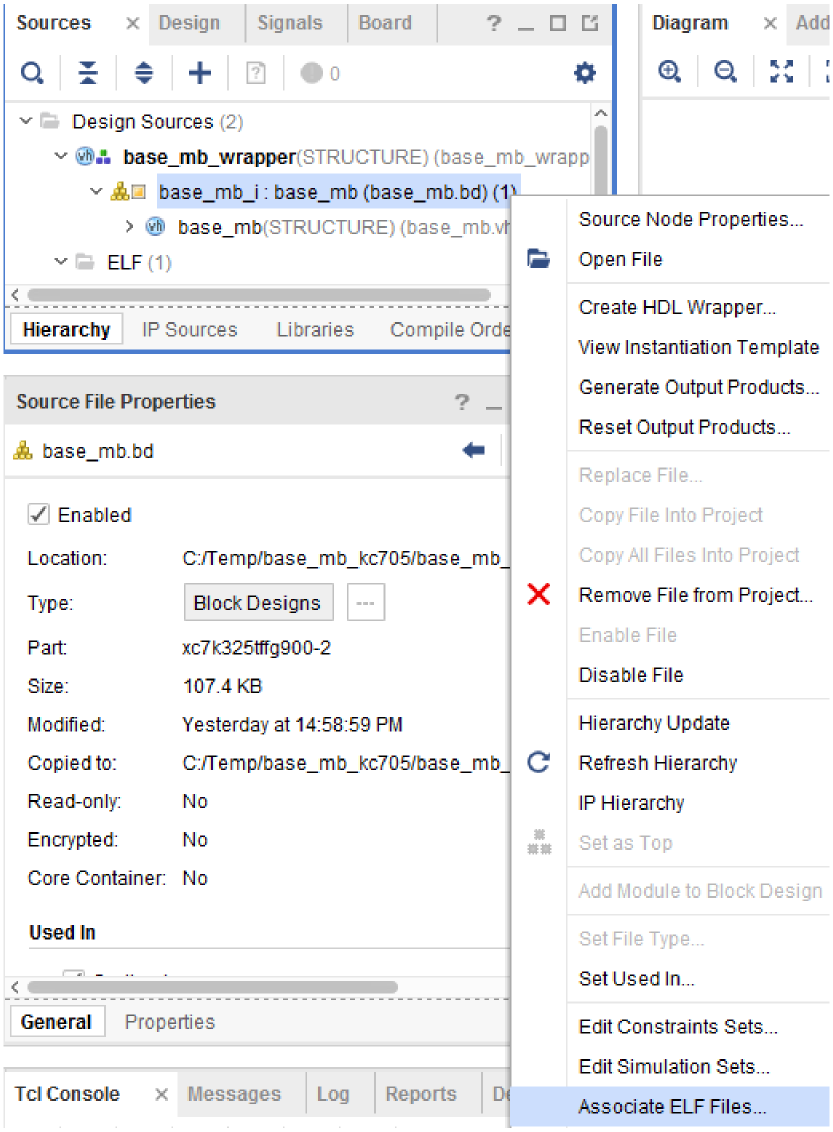
5. Select **Copy sources into project** to copy the ELF file into the local project, or leave the option unchecked to work with the original ELF file.
6. Click **Finish**.

   In the Sources window, you see the ELF file added under the ELF folder, as shown in the following figure.

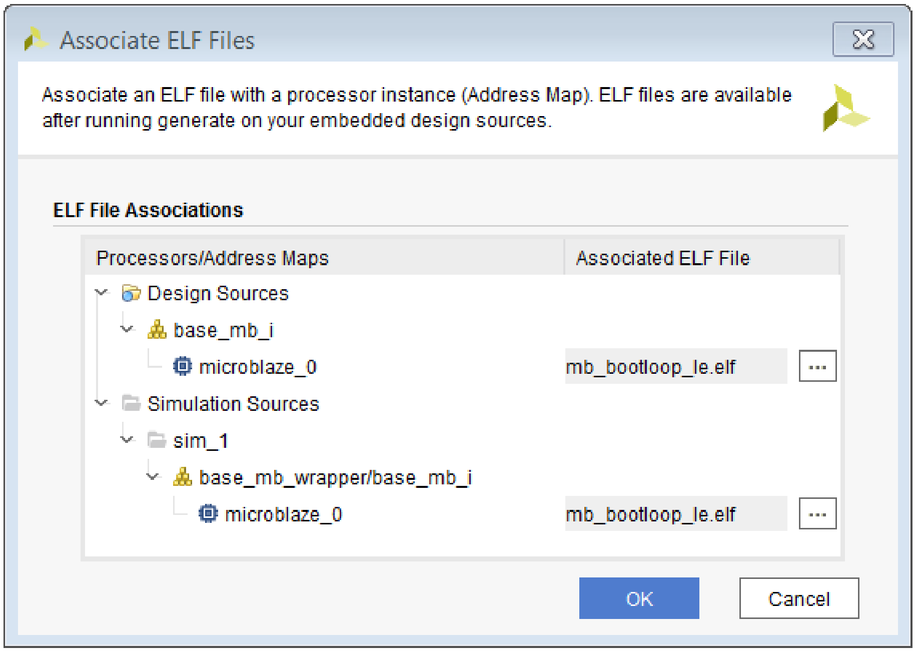


   After adding the ELF file to the project, you must associate the ELF file with the microprocessor in the design.

7. In the Sources window, right-click the block design, and select **Associate ELF Files**, as shown in the following figure.



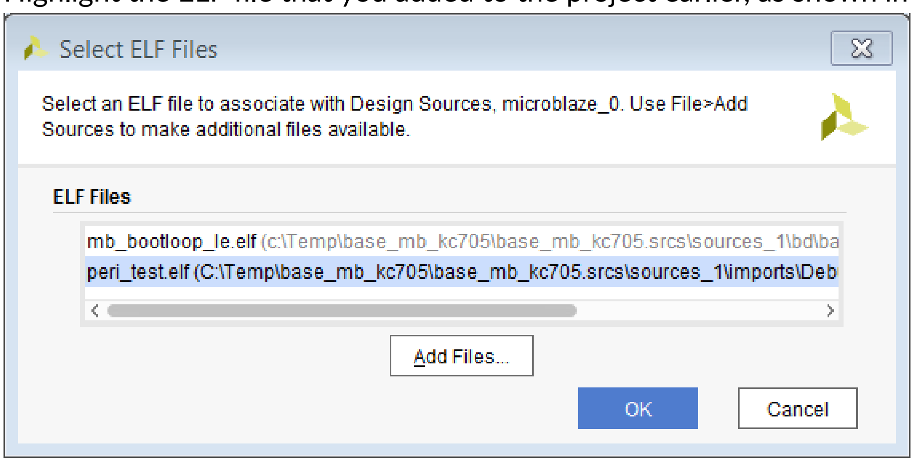
The Associate ELF File dialog box opens as shown in the following figure.



8. To associate an ELF as a design source for including in the bitstream, or as a source for use during simulation, click the appropriate Browse button.

   The Select ELF Files dialog box opens.

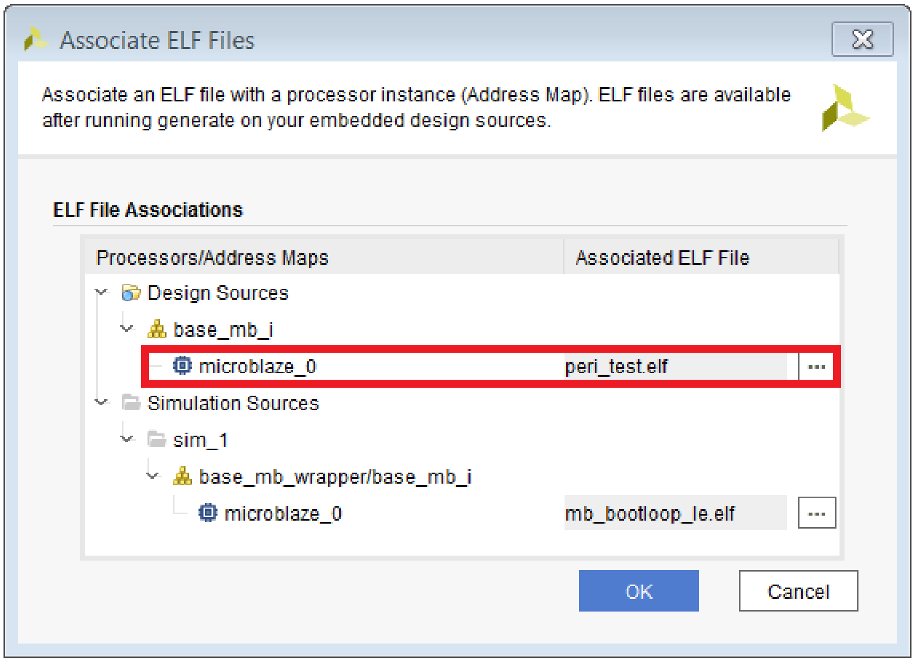
9. Highlight the ELF file that you added to the project earlier, as shown in the following figure.



TIP: *You can also click* ***Add Files*** *on the Select ELF Files dialog box to navigate to and add ELF files to the design at this time. In this case, the ELF file is referenced from its original location, and you do not have the option to copy it to the local project as you do if you add it using the* `Add Sources` *command.*

10. Ensure that the ELF file displays in the Associated ELF File column, as shown in the following figure, and click **OK**.

    With the ELF file added to the project, the Vivado tools automatically merge the Block RAM memory information (MMI file) and the ELF file contents with the device bitstream (BIT) when generating the bitstream to program the device.

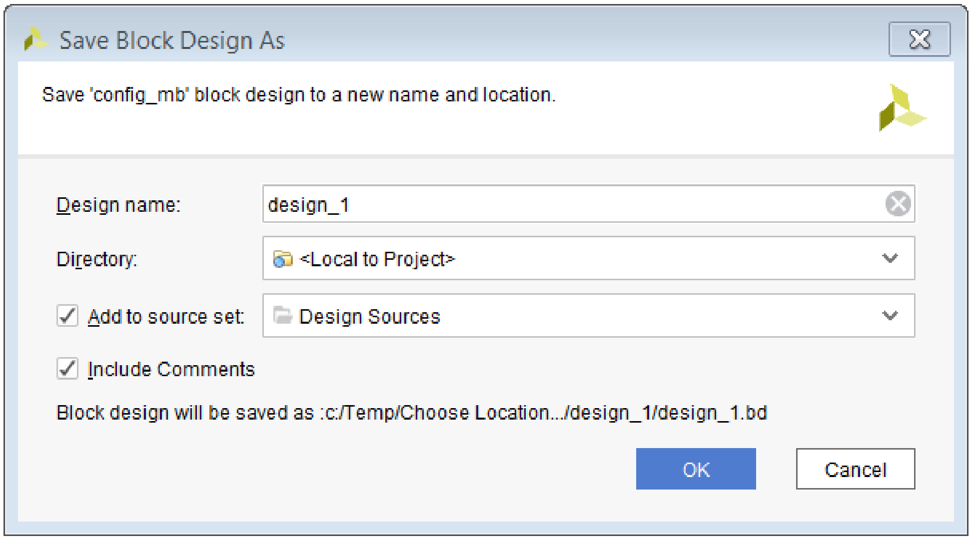


TIP: *You can also merge the MMI, ELF, and BIT files after the bitstream has been generated by using the update_mem utility. See this link in the Vivado Design Suite User Guide: Embedded Processor Hardware Design (UG898)) for more information.*

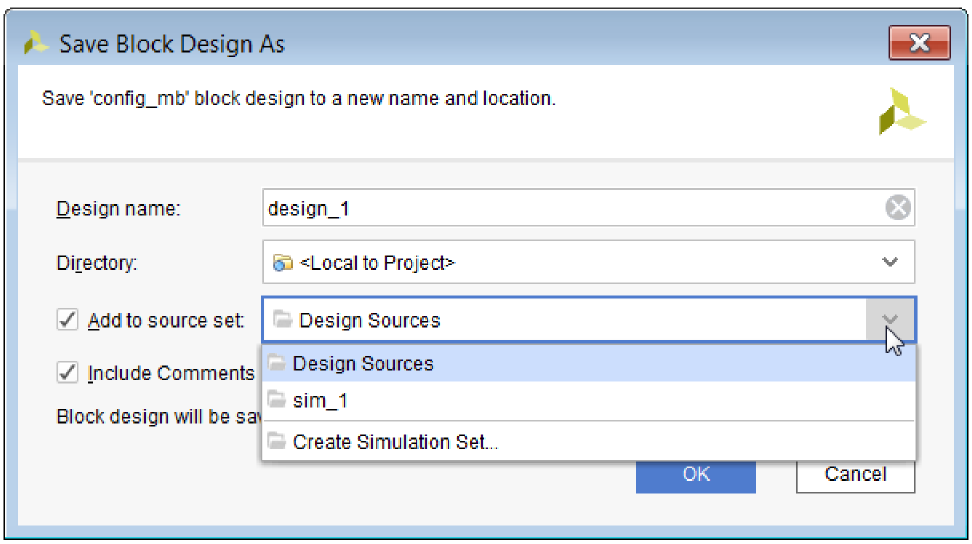
Saving a Block Design with a New Name

You might want to save a block design with a new name to add into another project or to create another copy of an existing block design to be added to the same project.

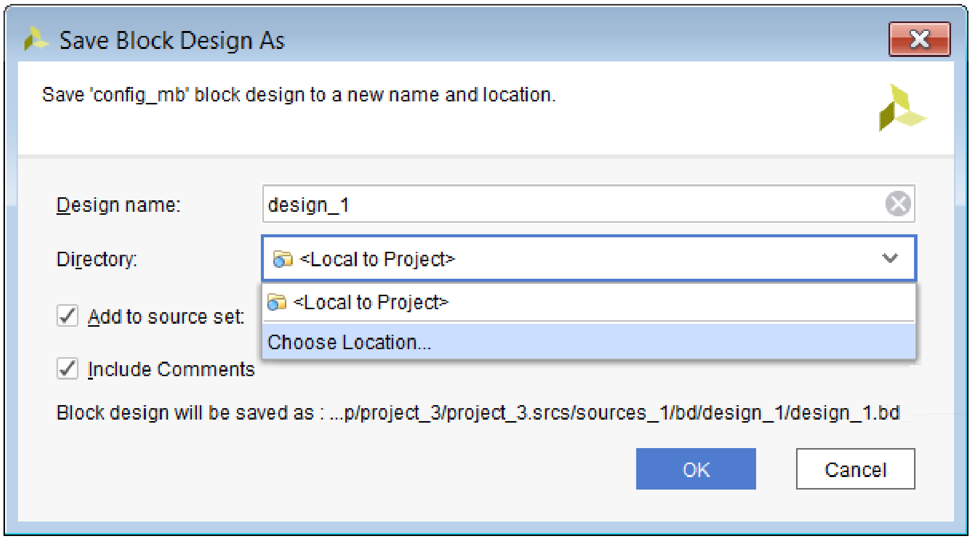
1. Select **File → Save Block Design As**.
2. Open the block design to save.
3. Decide where to save the block design. There are two options:
    - Create a copy of the block design and add it to the project sources. The default option for Directory is `<Local to Project>`.



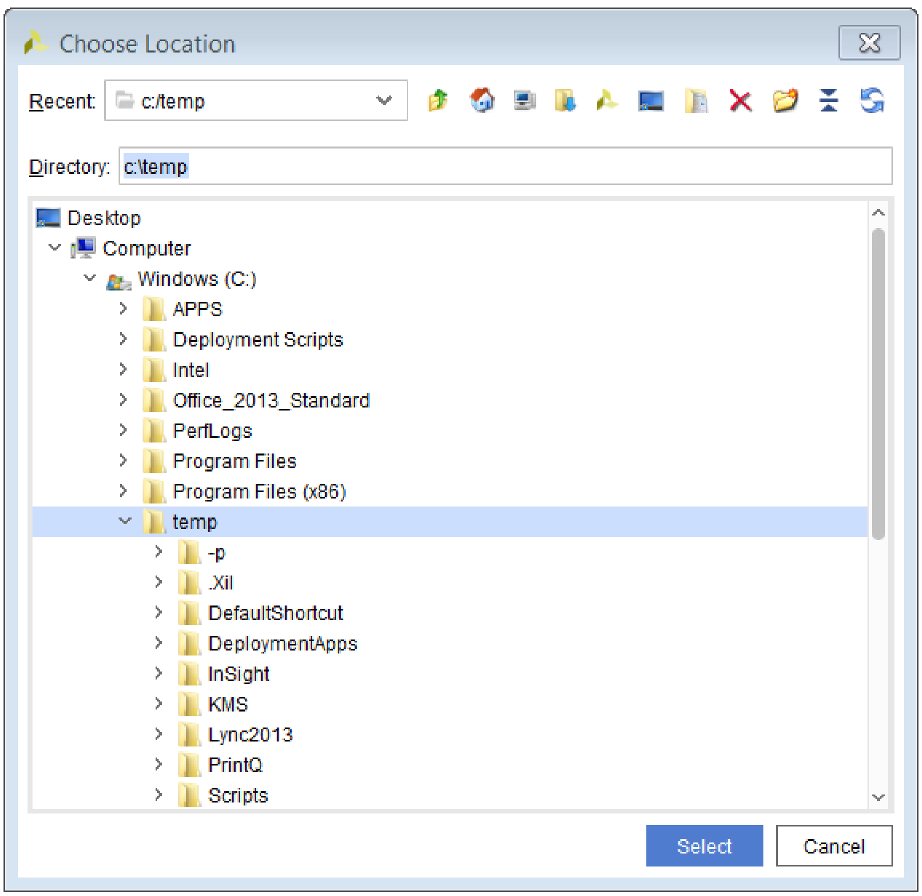
With this option selected, the block design is added to the project sources. The source set option can be set to Design Sources or Simulation Sources.



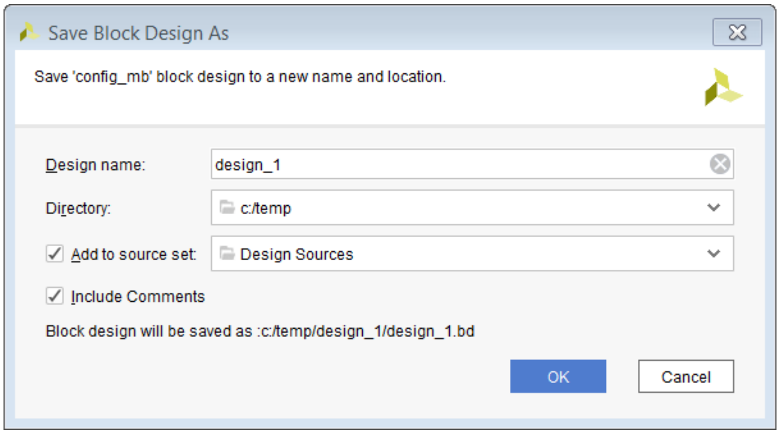
- Save the block design to a remote location.



The Choose Location dialog box opens.



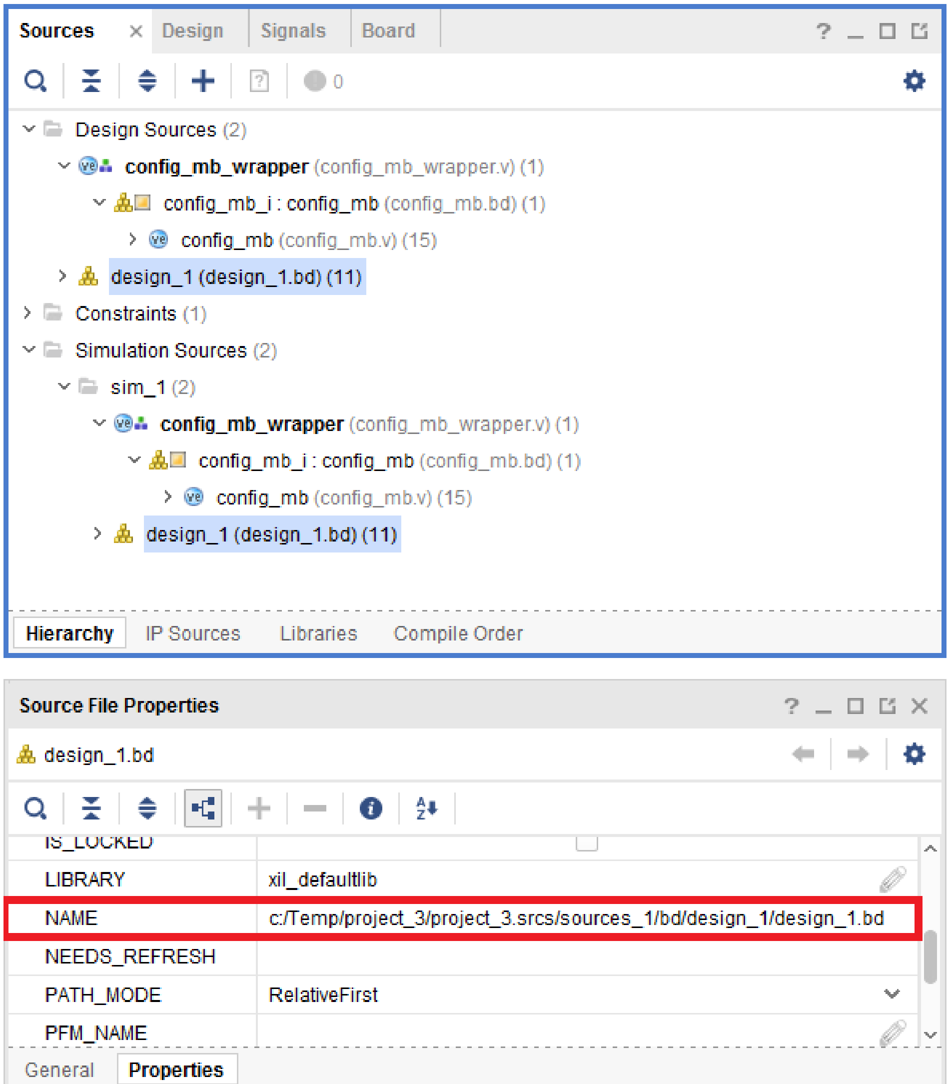
After you select the desired location, the Save Block Design as dialog box looks as follows:



Other options are exactly the same as saving the block design in the local project.

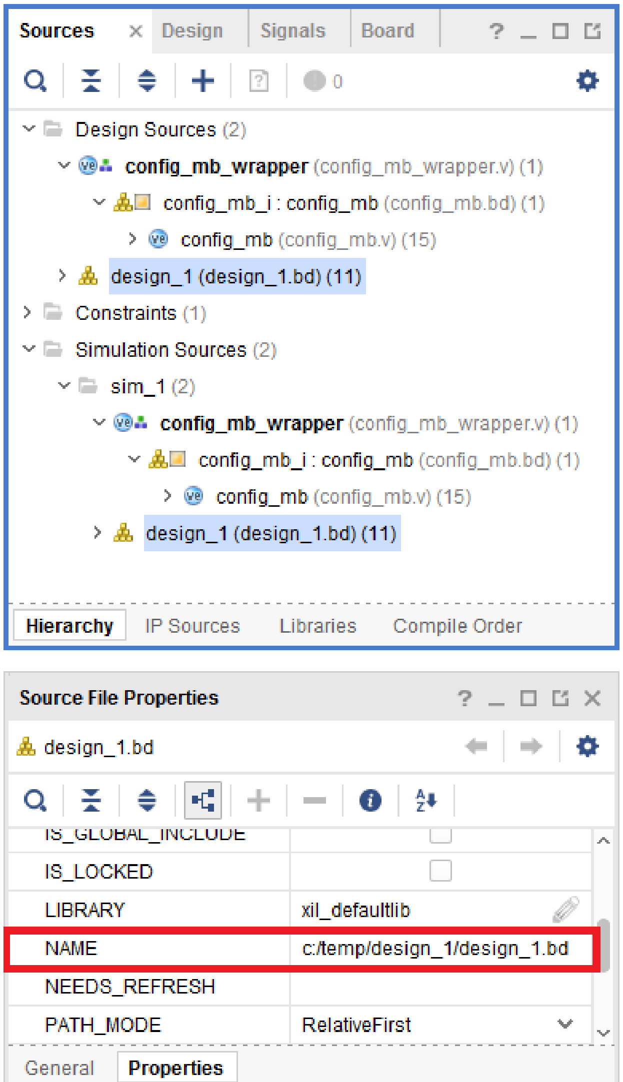
The Include Comments option preserves the comments in the original block design.

When the block design is saved locally, a copy of the block design appears in the sources directory as shown below.



***Note*:** The block design is copied local to the project, as can be seen in the Properties window.

When the block design is saved in a remote location (outside of the project), the newly-saved block design is added to the current project. The block design sources are saved in the remote location, and then referred to from the remote location where the block design exists, as can be seen in the Properties window, shown in the following figure.

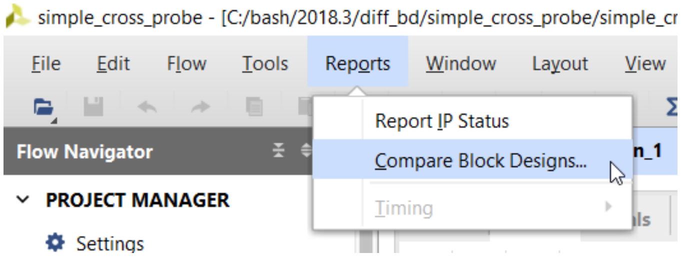


Comparing Two Block Designs

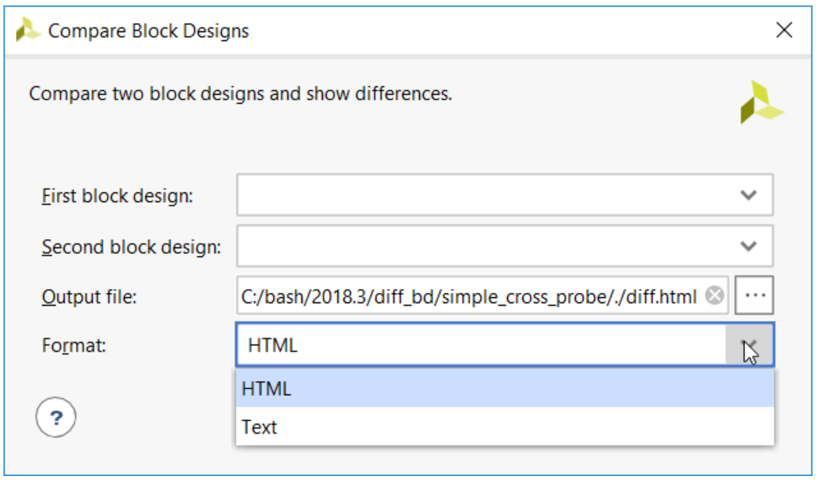
The Compare Block Designs command enables you to compare two block designs (BD), and generates a report showing the differences between the two BDs (in a diff report). This feature is useful in revision control systems to quickly compare and determine what has changed between two block designs; for example, between a block design that is under source control and a block design that you have checked out. The compare operation generates either a text report or an HTML report. If you choose HTML format, the generated report opens in a web browser window.

To compare two block designs:

1. Select **Reports → Compare Block Designs**, as shown in the following figure.

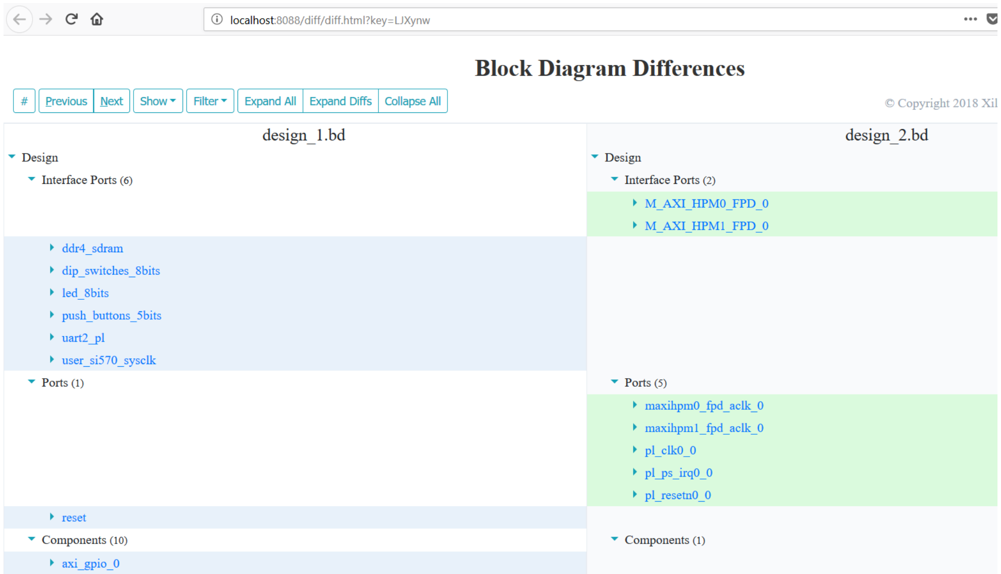


The Compare Block Design dialog box opens.



2. In the First block design drop-down field, specify the name of the first block design to compare. This can be in the local project directory or in a location outside of the current project.
3. In the Second block design drop-down field, specify the second block design file. This can be in the local project directory or in a location outside of the current project.
4. In the Output file field, you can leave the default value, or browse to a folder, and specify a name for the `diff` file.
5. From the Format drop-down menu, select **HTML** or **Text** for output file format.
6. Click **OK** to generate the file.

   If you selected HTML format, the generated HTML report file opens in the default web browser, as shown in the following figure.



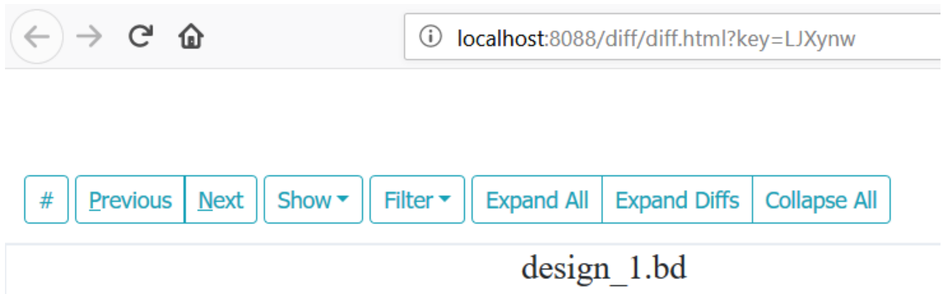
Note: Microsoft Internet Explorer and Edge web browsers do not open the HTML report by default. After the HTML report file is generated, you can manually open the report file in these web browsers.

If you selected Text format, the generated text report file opens in the text editor, as shown in the following figure.

Navigating in the HTML Diff Report

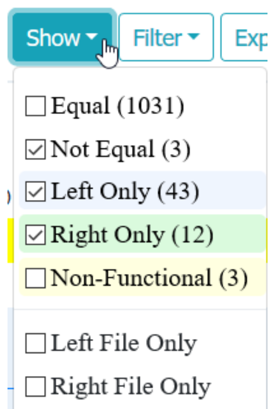
At the top of the HTML diff report, you will see menu items that enable you to navigate through the report.

Figure 111: **Menu Items for the diff Report**



- Click the # button to display line numbers in the report.
- Click Previous and Next to go to the previous or next difference in the report.
- The Show drop down list provides display customization options for you to select.

Figure 112: **Show Button Options**



- The Filter button drop down list provides content filtering options, such as Components, Connectivity, Parameters and/or Addressing. Click Advanced for additional filtering options.

Figure 113: **Filter Button Options**

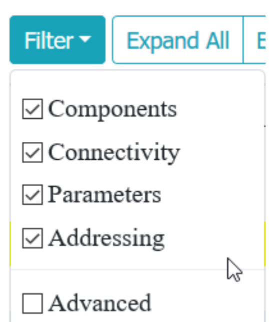
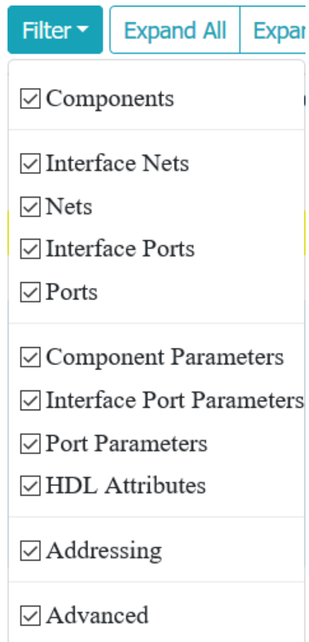


Figure 114: **Advanced Filter Button Options**



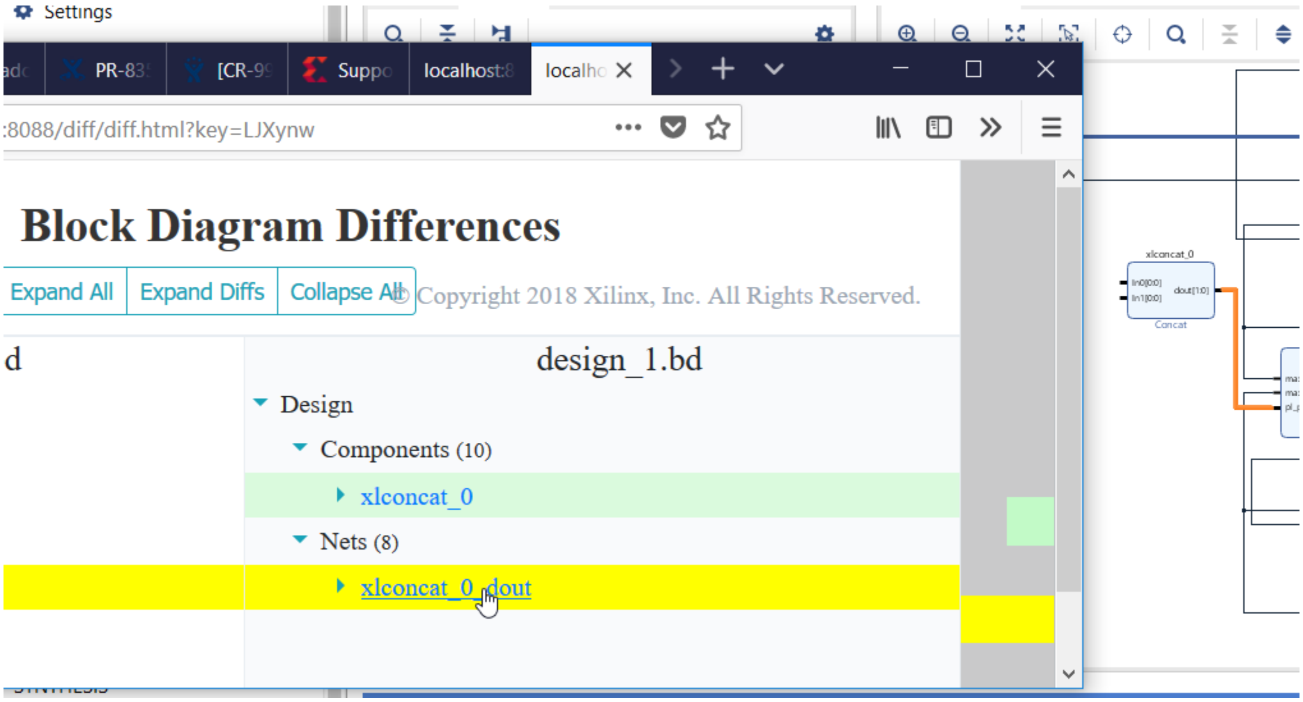
The differences are presented in a "collapsed" view in the report.

- You can expand all the items by clicking Expand All.
- Likewise, clicking Collapse All collapses all the differences.
- To expand only the items that differ, click Expand Diffs.

Cross-Probing Differences in the Block Designs

From the HTML diff report, you can cross-probe a reported difference and display the block design difference in Vivado. In the HTML report, the differences appear as hyperlinks. Click a hyperlink to highlight the item showing the difference. As an example, the following report shows a collapsed view of the differences between two BDs.

Figure 115: **Cross-probing**



In the above report, two differences are highlighted. For design_1.bd, under Components, you can see that a new component instance called xlconcat_0 is found. Under Nets, xlconct_0_dout is found.

- Click xlconcat_0 to highlight the `xlconcat_0` block on `design_1`.
- Click xlconct_0_dout to highlight the `xlconct_0_dout` net on `design_1`.

By expanding and then clicking on various differences in the report, you can see the differences between the two block designs being compared.

diffbd Command Line Utility

The standalone command called `diffbd` performs a non-graphical comparison of two block designs. This command returns a diff report for the two block designs specified. This command will return a zero if there are no functional differences between the BDs, and a one if there are differences. You can find out more information about this command by typing `diffbd -h` at the command line.

Block designs must be specified as BD objects, as returned by the `current_bd_design` or `get_bd_designs` command. The design objects can have the same name, but must be returned from different .bd files. An error is returned if the BD objects refer to the same design.

The differences reported include:

- Additions, or changes to the IP in use in the block diagram.
- Changes to design properties or parameters.
- Changes to the design hierarchy.
- Changes to connectivity.
- Changes to memory addressing.

Packaging a Block Design

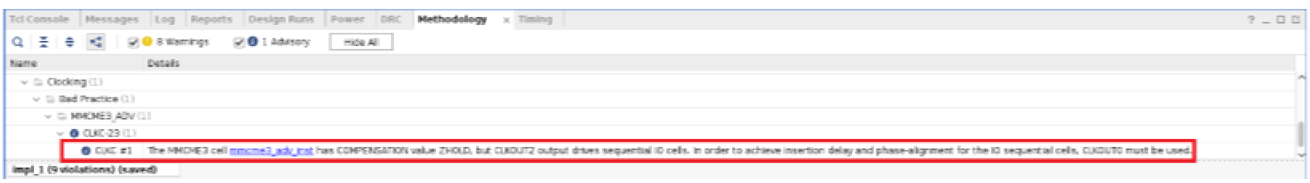
When you have created an IP integrator BD, implemented it, validated it, and tested it on the target hardware, and you are satisfied with the functionality of the BD, you can *package* the BD to create an IP that can be reused in another design.

For more information on packaging a BD for use in the Vivado IP catalog, see this link in the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

Cross-Probing Timing Paths

Often times there is a need to probe a timing issue after implementation, to a source block in IP Integrator design. This can be useful when a user is not intimately aware of a design inherited from another team member. To isolate a timing path to a particular block, the user can open up the implemented design and click on the link to point to the source block of the timing issue in question on the block design canvas. As an example, in the following implemented design, there is a methodology warning on an IP block.

Figure 116: **Methodology Warning**



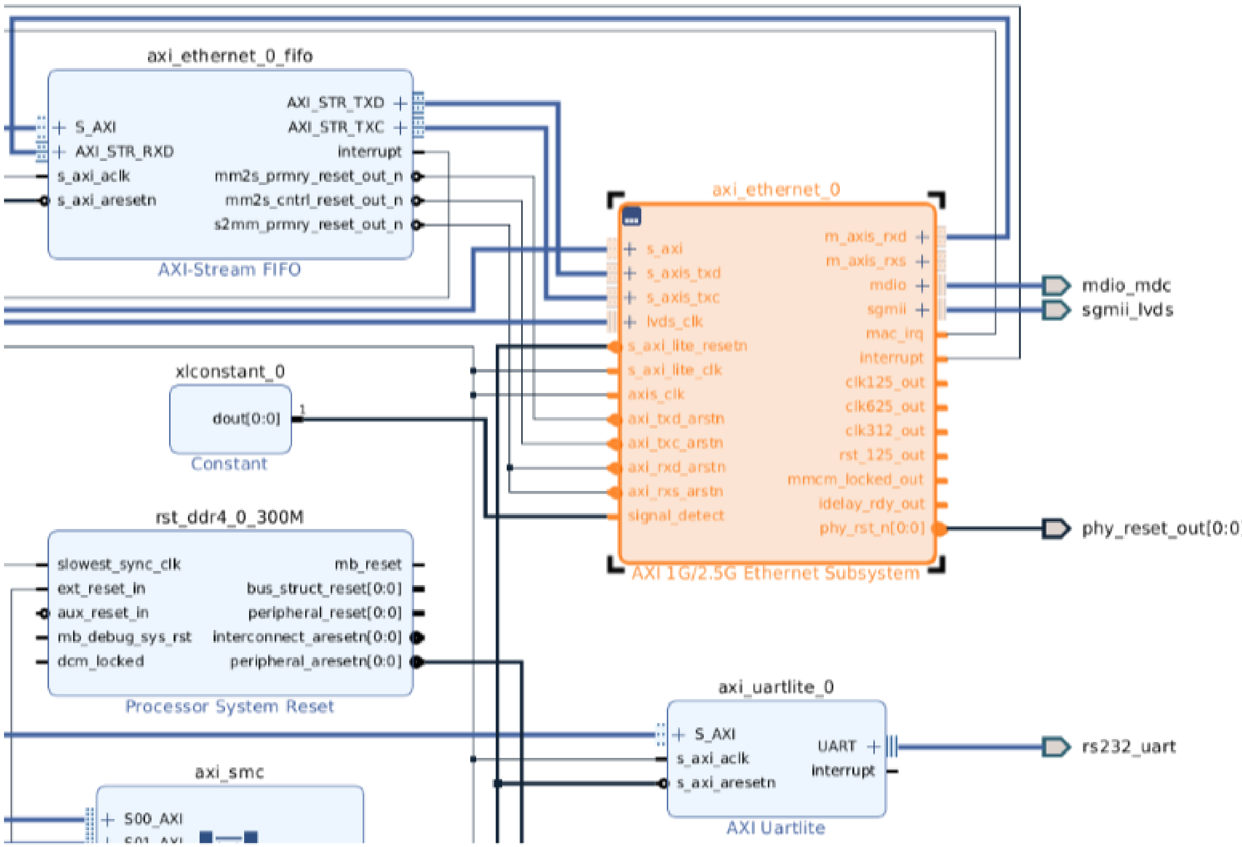
The message here says:

```
CLKC #1 The MMCME3 cell config_mb_i/axi_ethernet_0/inst/pcs_pma/inst/
core_clocking_i/mmcme3_adv_inst has COMPENSATION value ZHOLD, but CLKOUT2
output drives sequential IO cells. In order to achieve insertion delay and
phase-alignment for the IO sequential cells, CLKOUT0 must be used.
```

As you can see in the message the path cell `config_mb_i/axi_ethernet_0/inst/pcs_pma/inst/core_clocking_i/mmcme3_adv_inst` has a link. Clicking this link will take you to the block design canvas and highlight the block design cell related to this timing message.

To see the cell in question on the block design canvas, click **IP INTEGRATOR** in **Flow Navigator** to switch view to the Block Design Canvas.

Figure 117: **Highlighted cell in the message on the block design canvas**



Once the offending cell or IP block has been identified, the user can then look at the source code or the constraints file to identify the issue in hand.

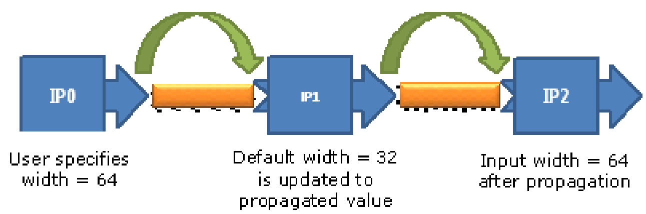
Chapter 6

Propagating Parameters in IP Integrator

Parameter propagation is one of the most powerful features available in IP integrator. The feature enables an IP to auto-update its parameterization based on how it is connected in the design. IP can be packaged with specific propagation rules, and IP integrator will run these rules as the diagram is generated.

For example, in the following figure, IP0 has a 64-bit wide data bus. IP1 is then added and connected, as is IP2.

Figure 118: **Parameter Propagation Concept**



In this case, IP2 has a default data bus width of 32 bits.

When you run the parameter propagation rules, you are alerted to the fact that IP2 has a different bus width. Assuming that the data bus width of IP2 can be changed through a change of parameter, IP integrator can automatically update IP2.

If the IP cannot be updated to match properties based on its connection, an error displays, alerting you of potential issues in the design. This simple example demonstrates the power of parameter propagation. The types of errors that can be corrected or identified by parameter propagation are often errors not found until simulation.

Using Bus Interfaces

A bus interface is a grouping of signals that share a common function. An AXI4-Lite master, for example, contains a large number of individual signals plus multiple buses, which are all required to make a connection.

One of the important features of IP integrator is the ability to connect a logical group of bus interfaces from one IP to another, or from the IP to the boundary of the IP integrator design or even the FPGA I/O boundary. Without the signals being packaged as a bus interface, the symbol for the IP shows an extremely long and unusable list of low-level ports, which are difficult to connect one-by-one.

A list of signals can be grouped in IP-XACT using the concept of a bus interface with its constituent port map that maps the physical port (available on the RTL or the netlist of the IP) to a logical port as defined in the IP-XACT abstraction definition file for that interface type.

Common Internal Bus Interfaces

Some common examples of bus interfaces are buses that conform to the AXI specification such as AXI4, AXI4-Lite and AXI4-Stream. The `AXIMM` interface includes all three subsets (AXI4, AXI3, and AXI4-Lite). Other interfaces include block RAM.

I/O Bus Interfaces

Some bus interfaces that group a set of signals going to I/O ports are called I/O interfaces. Examples include: UART, I2C, SPI, Ethernet, PCIe, and DDR.

Special Signals

Special signals include:

- Clock
- Reset
- Interrupt
- Clock Enable
- Data (for traditional arithmetic IP which do not have any AXI interface, for example adders, subtractors, and multipliers)

These special signals are described in the following sections.

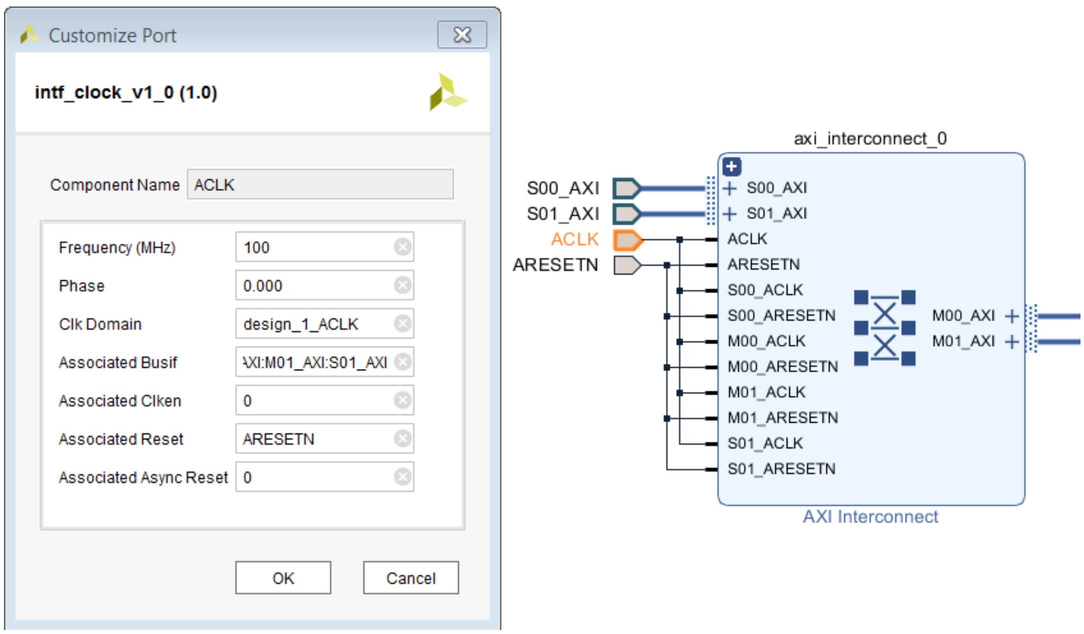
Clock

The clock interface can have the following parameters associated with them. These parameters are used in the design generation process and are necessary when the IP is used with other IP in the design.

- ASSOCIATED_BUSIF: The list contains the names of all bus interfaces which run at this clock frequency. This parameter takes a colon-separated list (:) of strings as its value.

If there are no interface signals at the boundary that run at this clock rate, this field is left blank.

Figure 119: **ASSOCIATED_BUSIF**



The figure shows the ASSOCIATED_BUSIF parameter of the selected clock interface port lists the master interfaces (M00_AXI and M01_AXI) and slave interfaces (S00_AXI and S01_AXI) separated by colons.

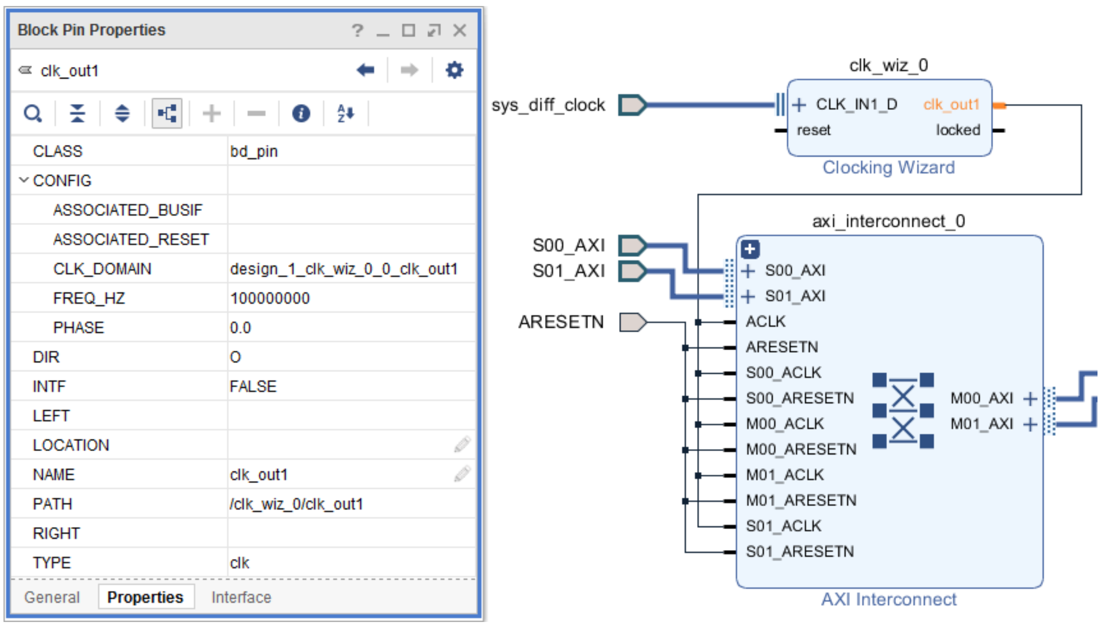
If one of the interfaces, such as M00_AXI, does not run at this clock frequency, leave the interface out of the ASSOCIATED_BUSIF parameter for the clock.

- ASSOCIATED_RESET: The list contains names of reset ports (not names of reset container interfaces) as its value. This parameter takes a colon-separated (:) list of strings as its value. If there are no resets in the design, this field is left blank.
- ASSOCIATED_CLKEN: The list contains names of clock enable ports (not names of container interfaces) as its value. This parameter takes a colon-separated (:) list of strings as its value. If there are no clock enable signals in the design, this field is left blank.
- FREQ_HZ: This parameter captures the frequency in hertz at which the clock is running in positive integer format. This parameter needs to be specified for all output clocks only.

- `PHASE`: This parameter captures the phase at which the clock is running. The default value is 0. Valid values are 0 to 360. If you cannot specify the `PHASE` in a fixed manner, then you must update it in bd.tcl, similar to updating `FREQ_HZ`.
- `CLK_DOMAIN`: This parameter is a string ID. By default, IP integrator assumes that all output clocks are independent and assigns a unique ID to all clock outputs across the block design. This is automatically assigned by IP integrator, or managed by IP if there are multiple output clocks of the same domain.

To see the properties on the clock net, select the source clock port or pin and analyze the properties on the object. The following figure shows the Clocking Wizard and the clock properties on the selected pin.

Figure 120: **Clock Properties**



You can use the report_property Tcl command, as follows:

```
report_property [get_bd_pins clk_wiz_0/clk_out1]
```

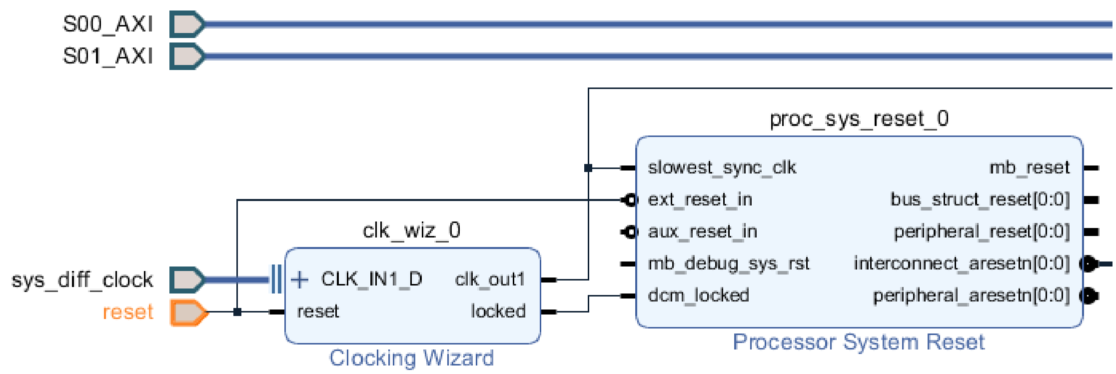
You can also double-click a port or pin to see the customization dialog box for the selected object.

Reset

This container bus interface includes the `POLARITY` parameter. Valid values for this parameter are `ACTIVE_HIGH` or `ACTIVE_LOW`. The default is `ACTIVE_LOW`.

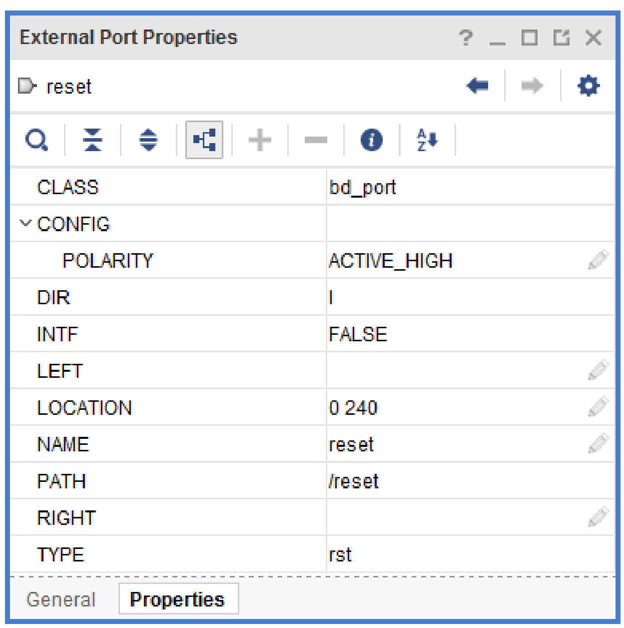
To see the properties on the reset net, select the reset port or pin and analyze the properties on the object, as shown in the following figure.

Figure 121: **Reset Signal**



The following figure shows the Properties window.

Figure 122: **Reset Properties Window**

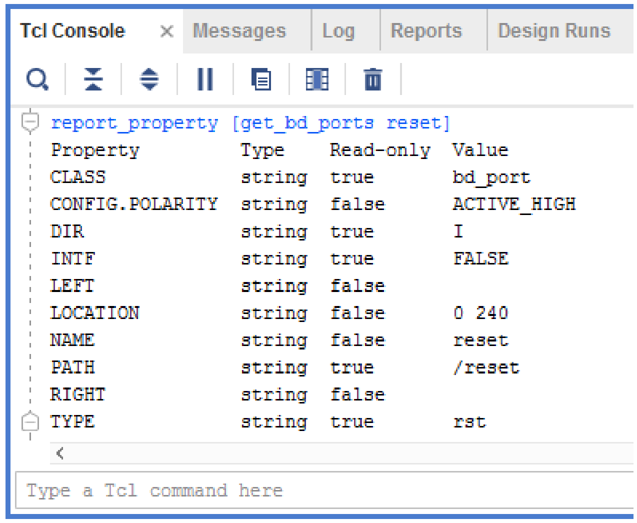


You can use the report_property Tcl command, as follows:

```
report_property [get_bd_ports reset]
```

This command writes the following output to the Tcl Console.

Figure 123: **Properties of the Reset Port**

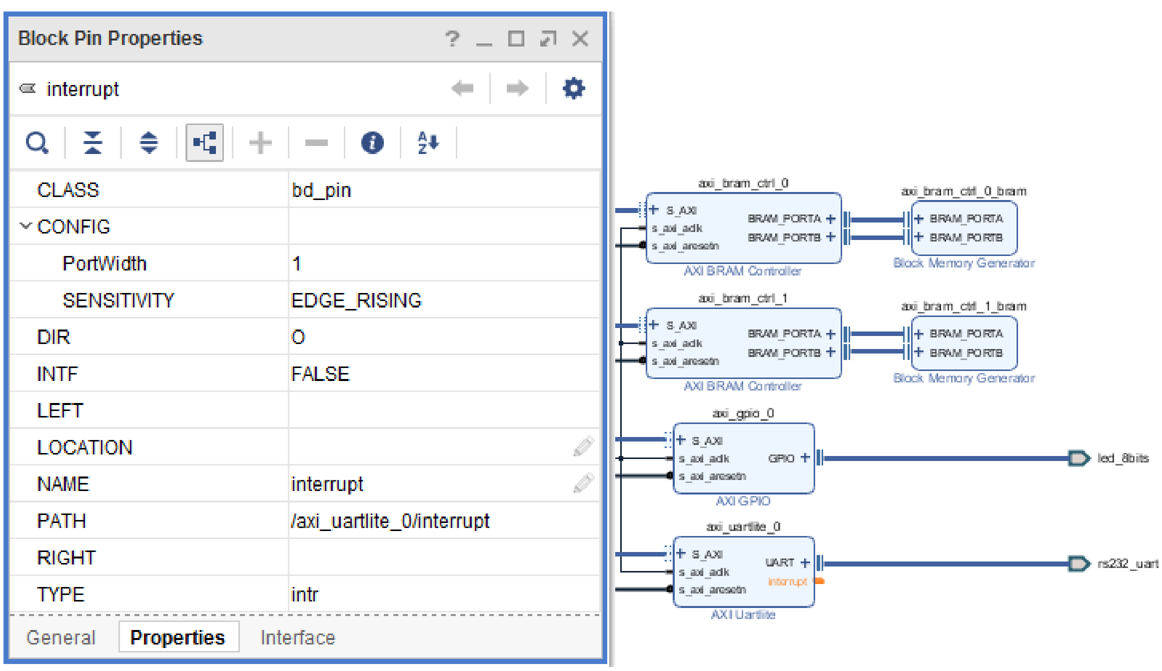


Interrupt

This bus interface includes the parameter, `SENSITIVITY`: Valid values for this parameter are `LEVEL_HIGH`, `LEVEL_LOW`, `EDGE_RISING`, and `EDGE_FALLING`. The default is `LEVEL_HIGH`.

To see the properties on the interrupt pin, highlight the pin, and look at the properties window, as shown in the following figure.

Figure 124: **Interrupt Properties: Block Diagram and Properties Window**

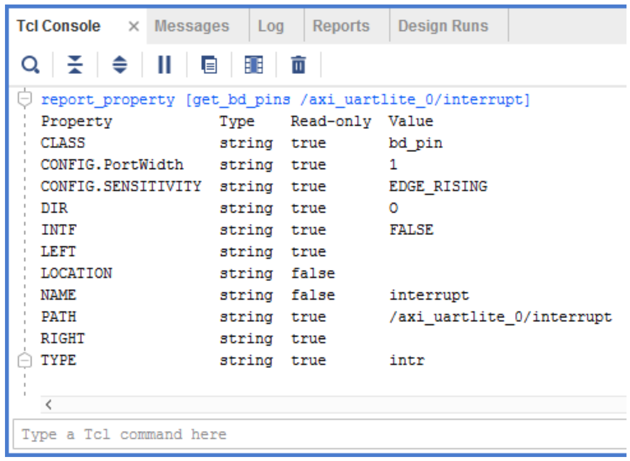


You can use the report_property Tcl command, as follows:

```
report_property [get_bd_pins /axi_uartlite_0/interrupt]
```

This command returns the information shown in the following figure.

Figure 125: **Reporting Interrupt Properties**



Clock Enable

There are two parameters associated with Clock Enable: `FREQ_HZ` and `PHASE`.

How Parameter Propagation Works

In IP integrator, parameter propagation takes place when you choose to run Validate Design. You can do this in one of the following ways:

- Click Validate Design in the Vivado® IDE toolbar.
- Click Validate Design button in the design canvas toolbar, or press F6.
- Select Tools > Validate Design from the Vivado menu.
- Use the Tcl command: validate_bd_design at the Tcl Console.

Parameter propagation synchronizes the configuration of an IP instance with that of other instances to which it is connected. The synchronization of configuration happens at bus interface parameters.

The parameter propagation in the IP integrator works primarily on the concept of assignment strength for an interface parameter. An interface parameter can have a strength of `USER`, `CONSTANT`, `PROPAGATED`, or `DEFAULT`. When the tool compares parameters across a connection, it always copies a parameter with higher strength to a parameter with lower strength.

Parameters in the Customization GUI

In the Non-Project Mode, you must configure all user parameters of an IP. In the context of IP integrator, any user parameters that are auto-updated by parameter propagation are grayed out in the IP customization dialog box. A grayed-out parameter indicates that you cannot set the specific-user parameters directly on the IP; instead, the property values are auto-computed by the tool.

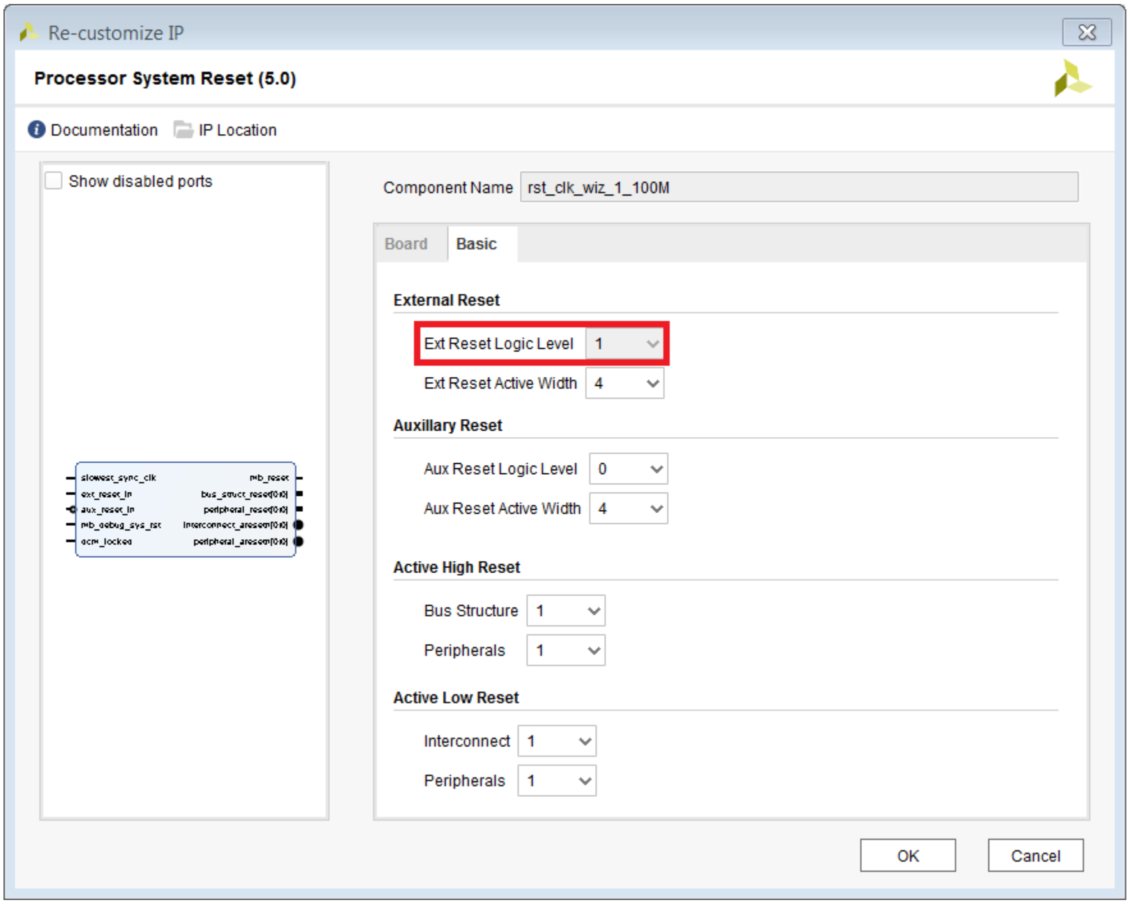
There are situations when the auto-computed values might not be optimal. In those circumstances, you may override these propagated values.

The cases in which you encounter parameter propagation are, as follows:

- Auto-computed parameters: Parameters are auto-computed by the IP integrator and you cannot override them. For example, the Ext Reset Logic Level parameter in the following figure is gray to indicate you cannot change this parameter.

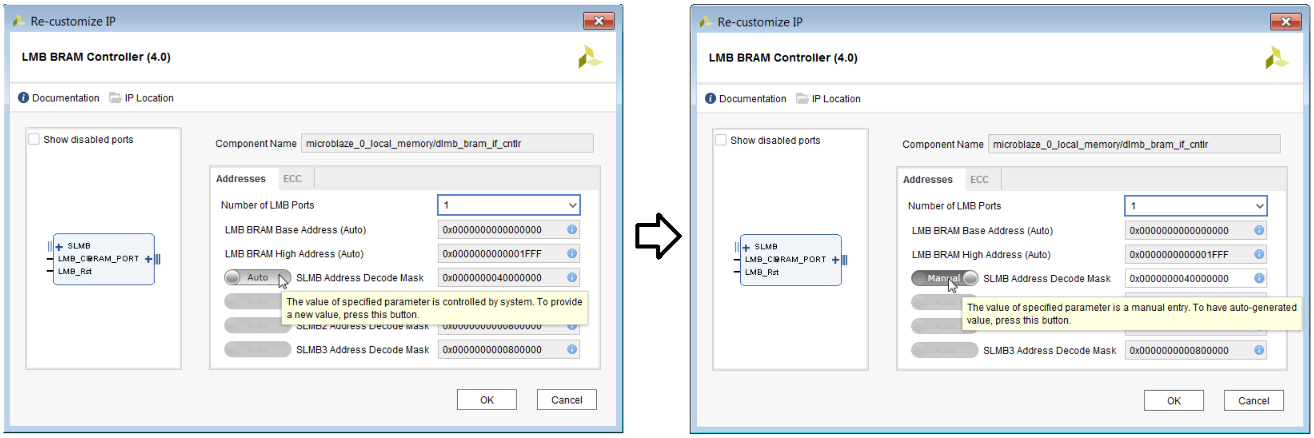
The following figure shows the Re-customize IP pane of the Processor System Reset.

Figure 126: **Auto-Computed Parameter**



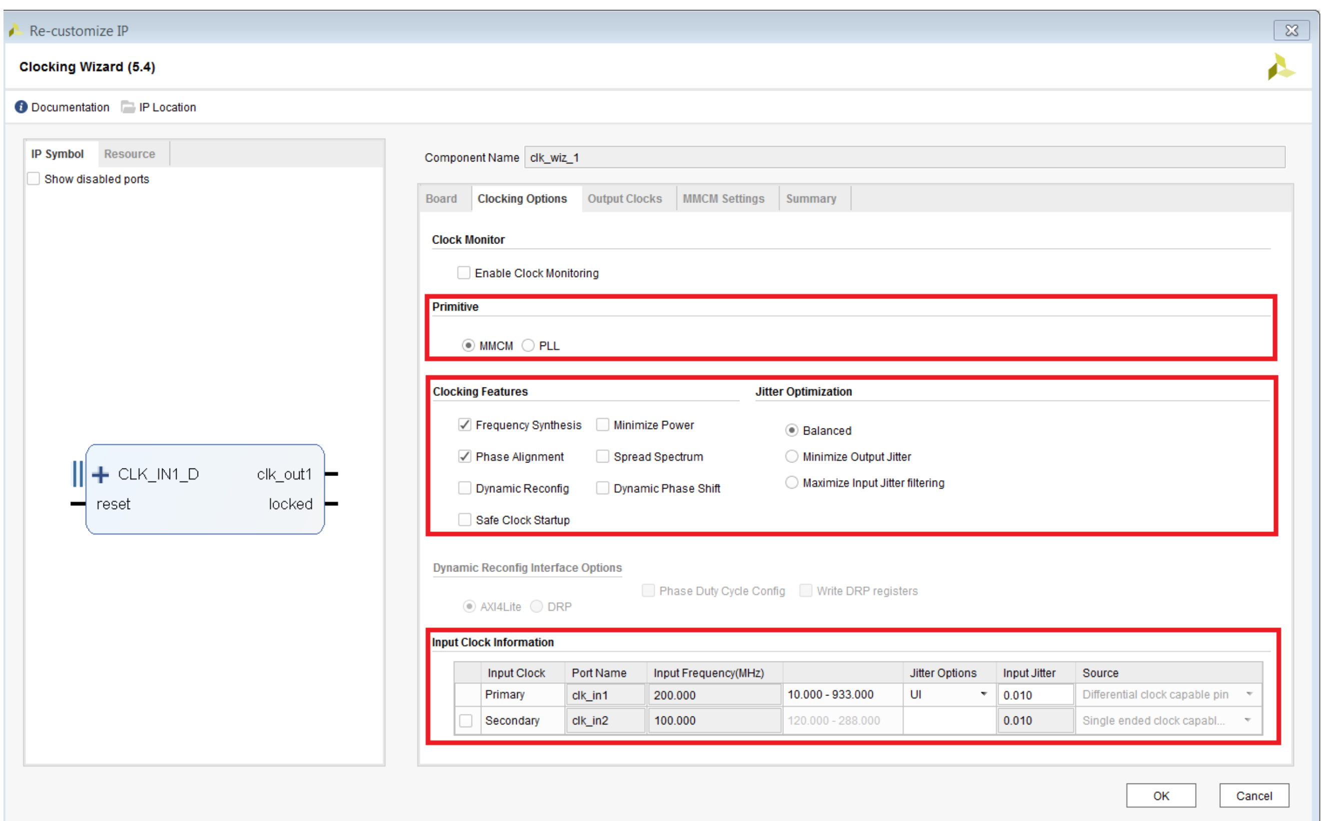
- Override-able parameters: Auto-computed parameters that you can override. For example, you can change the SLMB Address Decode Mask for the LMB BRAM Controller. When you hover the mouse on top of the slider button, it informs you that the parameter is controlled by the system; but, you can change it by toggling the button from Auto to Manual. The following figure shows these settings.

Figure 127: **Parameter to Override**



- User configurable parameters: User configurable only. The following figure shows such parameters outlined in red.

Figure 128: **User-Configurable Parameter**

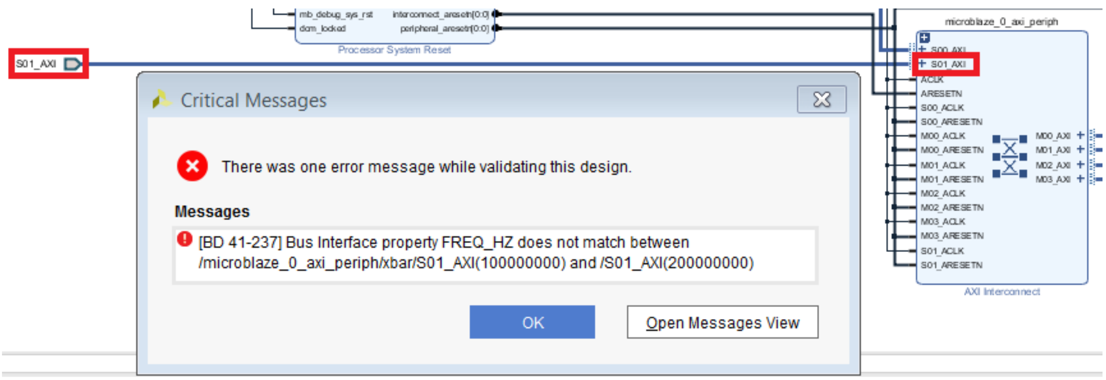


- Constants: Parameters that cannot be set.

Parameter Mismatch Example

The following is an example of a parameter mismatch on the `FREQ_HZ` property of a clock pin. In this example, the frequency does not match between the `S01_AXI` port and the `S_AXI` interface of the AXI Interconnect. This error is revealed when the design is validated.

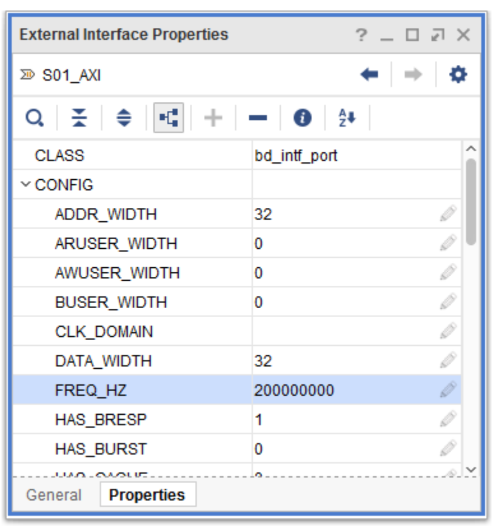
Figure 129: **FREQ_HZ Property Mismatch Between Port and Interface Pin**



- The `S01_AXI` port has a frequency of 200 MHz as can be seen in the properties window.
- The `S01_AXI` interface of the AXI Interconnect is set to a frequency of 100 MHz.

You can fix this error by changing the frequency in the property, or by double-clicking the `S01_AXI` port and correcting the frequency in the Frequency field of the customization dialog box.

Figure 130: **Change Frequency Port in Properties Window**



After you change the frequency, re-validate the design to ensure there are no errors.

Chapter 7

Debugging IP Integrator Designs

In-system debugging lets you debug your design in real-time on your target hardware. This is an essential step in design completion. Invariably, one comes across a situation which is extremely hard to replicate in a simulator. Therefore, there is a need to debug the problem in the FPGA. In this step, you place an instrument into your design with special debugging hardware to provide you with the ability to observe and control the design. After the debugging process is complete, you can remove the instrumentation or special hardware to increase performance and reduce logic.

The Vivado® IP integrator provides ways to instrument your design for debugging which is explained in the following sections:

- Using the HDL Instantiation Flow in IP Integrator
- Using the Netlist Insertion Flow

Choosing the best flow for debugging your block design depends on your preference and the types of nets and signals that you want to debug.

As an example:

- If you are interested in performing hardware-software co-verification using the cross-trigger feature of a MicroBlaze or Zynq®-7000 processor, you can use the HDL Instantiation flow.
- If you are interested in verifying interface level connectivity, then you can use the HDL Instantiation flow.
- If you are interested in debugging the post implemented design, you can use the Netlist Insertion flow or the HDL Instantiation flow.

You can also use a combination of both flows to debug the block design and the top-level design.

***Note*:** See the Vivado Design Suite QuickTake Video: AXI Interface Debug Using IP Integrator for information on debugging an AXI interface.

Using the HDL Instantiation Flow in IP Integrator

For debugging the elements of a block design using the Vivado Hardware Manager, the IP integrator provides two distinct IP cores:

- Integrated Logic Analyzer (ILA): This is a legacy debug core for block designs, *that is no longer recommended for use*. The Integrated Logic Analyzer (ILA) debug core lets you perform in-system debugging of implemented block designs to monitor signals in the design, to trigger on hardware events, and to capture data at system speeds. Detailed documentation on the ILA debug core can be found in the *Integrated Logic Analyzer LogiCORE IP Product Guide* (PG172).
- System ILA: The System Integrated Logic Analyzer (System ILA) debug core is a logic analyzer that lets you monitor interfaces and signals in IP integrator block design, to trigger on interface and signal related hardware events, and to capture data at system speeds. The System ILA debug core offers AXI interface debug and monitoring capability along with AXI4-MM and AXI4-Stream protocol checking.

The System ILA core is synchronous to the nets being monitored or debugged, so all design clock constraints applied to that particular clock domain are also applied to the components of the System ILA core. Detailed documentation on the System ILA core IP can be found in the *System Integrated Logic Analyzer LogiCORE IP Product Guide* (PG261).

***Note*:** Existing block designs can continue to use the ILA debug core. However, new block designs should use the new System ILA debug core to take advantage of the advanced features and ease-of-use of this core.

Using the System ILA IP to Debug a Block Design

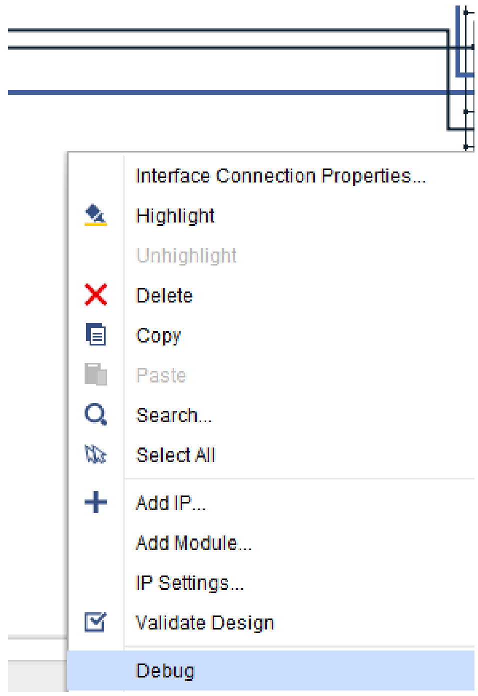
The System ILA debug core in IP integrator allows you to perform in-system debugging of block design on a Xilinx® device. This feature should be used when there is a need to monitor interfaces and signals in the design.

The IP integrator debugging flow has four distinct phases:

1. Mark the interfaces or nets to be probed using the Debug option.
2. Use Designer Assistance to connect the interfaces and nets to the System ILA core.
3. Validate Design to ensure that design connectivity is correct.
4. Implement the design, and debug the design on hardware using the Vivado Hardware Manager.

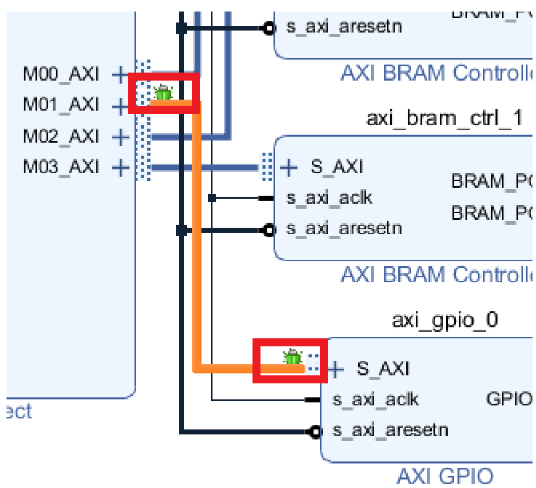
Nets can be marked for debug in the block design by right-clicking on the net and selecting Debug from the context menu as shown in the following figure.

Figure 131: **Mark Nets to Debug from Context Menu**



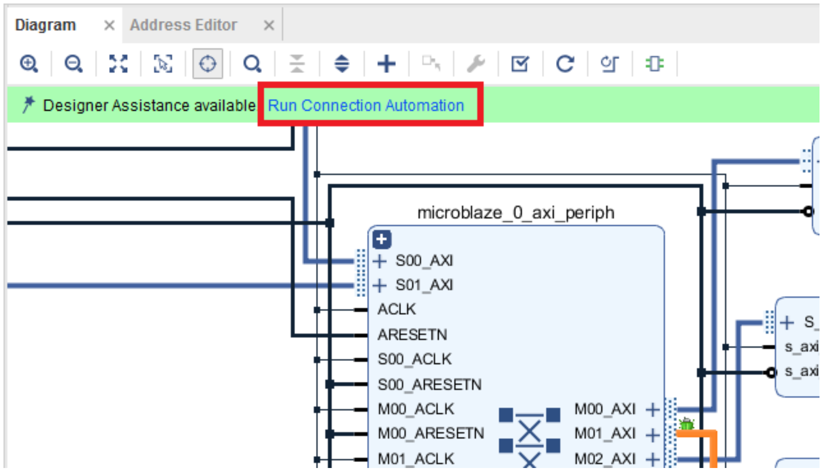
The nets that are marked for debug show a small bug icon placed on top of the net in the block design.

Figure 132: **Bug Icons on Nets to be Debugged**



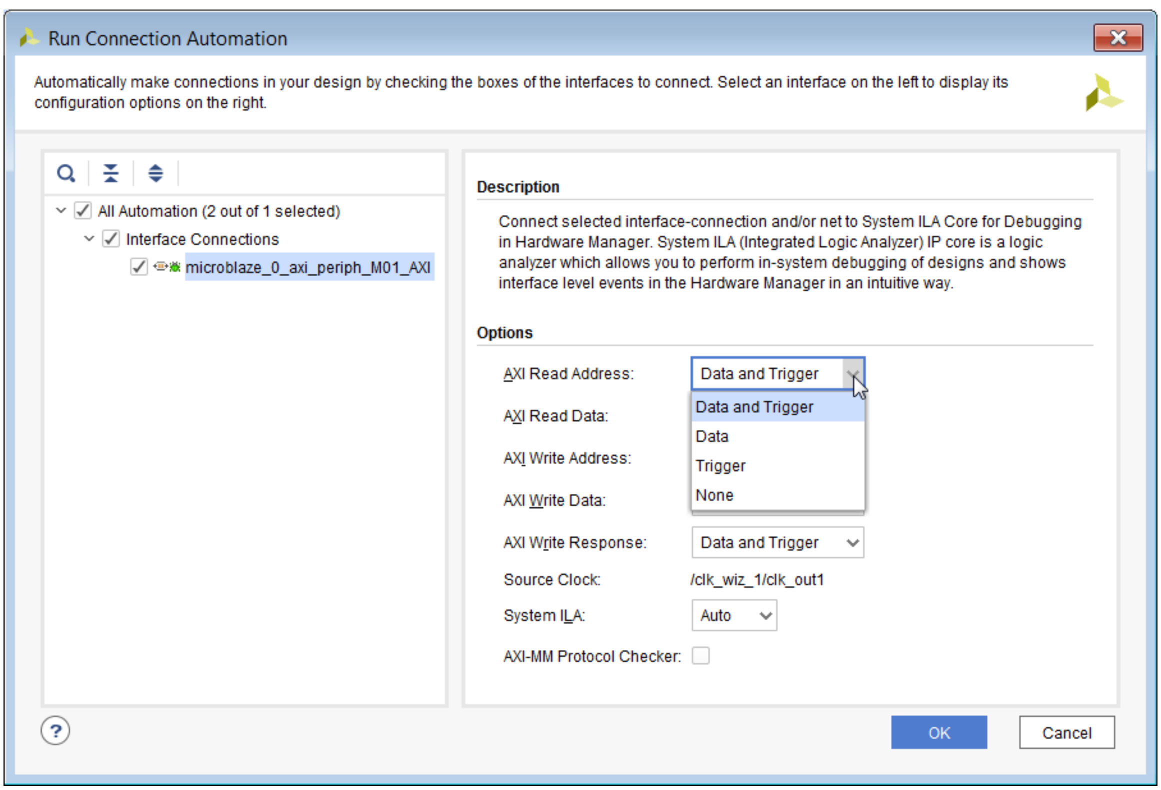
Note that the Run Connection Automation link is active in the block design canvas banner.

Figure 133: **Run Connection Automation to Connect Nets to be Debugged to System ILA**



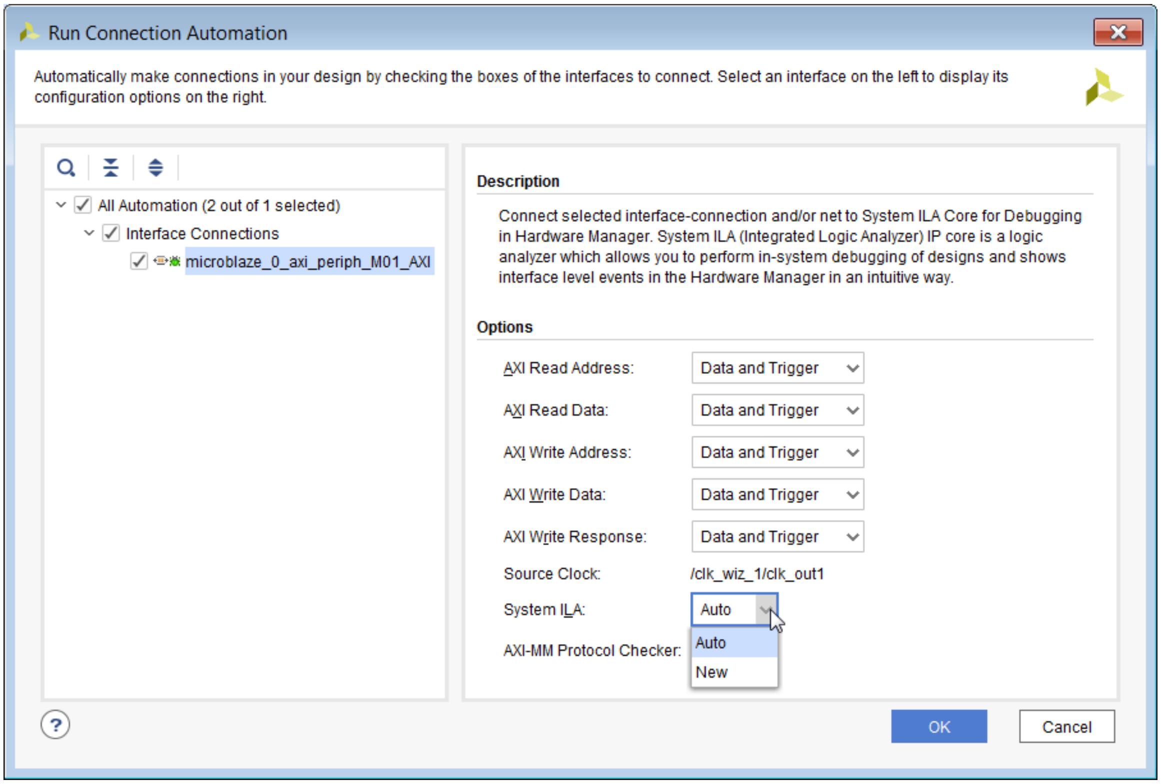
Clicking the Run Connection Automation link displays the Run Connection Automation dialog box, a provides the Run Connection Automation options shown in the following figure.

Figure 134: **Selecting Data and/or Trigger Option for Interface Signals**



Because the net being debugged in this case is an AXI Interface, interface pins such as Read/Write address and data pins are presented for setting Data and/or Trigger options. Similar options to set Data/Trigger options are presented when you mark a non-interface net is for debug and click the Run Connection Automation link.

Figure 135: **Setting System ILA Options**



As shown, the System ILA option provides the user with two separate options:

- Auto: Lets the tool determine whether a new System ILA debug core should be used, or if the selected signals can be connected to an existing System ILA.
- New: Specifically connects the selected debug signals to a new System ILA IP core. In some cases this may be desired to keep certain signals connected to a particular ILA.

When no System ILA are present in the block design, choosing either option will instantiate a new debug core. The clock domain of the net being debugged is determined by the tool and is connected to the clk pin of the System ILA IP. If nets to be debugged are in different clock domains, separate System ILA debug cores are instantiated as it can only be connected to one clock source.

The Run Connection Automation dialog box also provides you with the option to connect the interface to an AXI Memory Mapped Protocol Checker, as shown in the following figure. The AXI Protocol Checker monitors AXI interfaces. When attached to an interface, it actively checks for protocol violations and provides an indication of which violation occurred.

TIP: *Additional details of debugging AXI interfaces in the Vivado Hardware Manager are described at this link in the Vivado Design Suite User Guide: Programming and Debugging (UG908).*

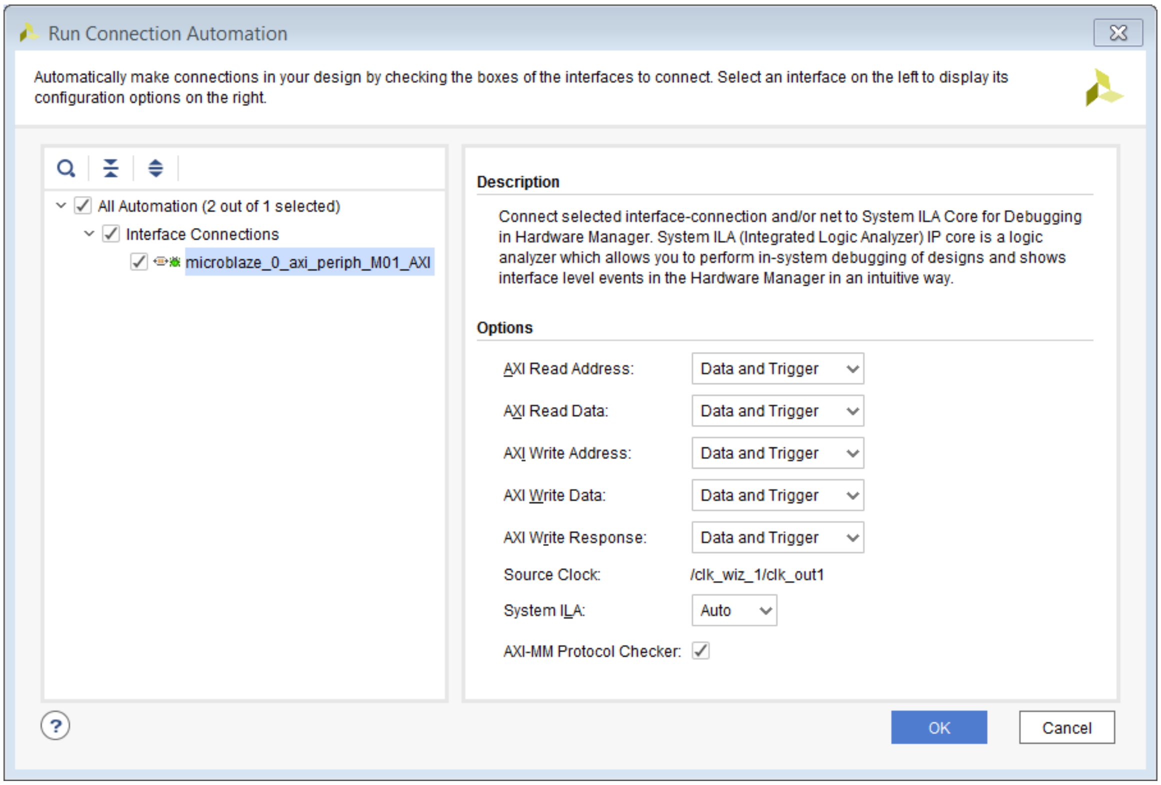


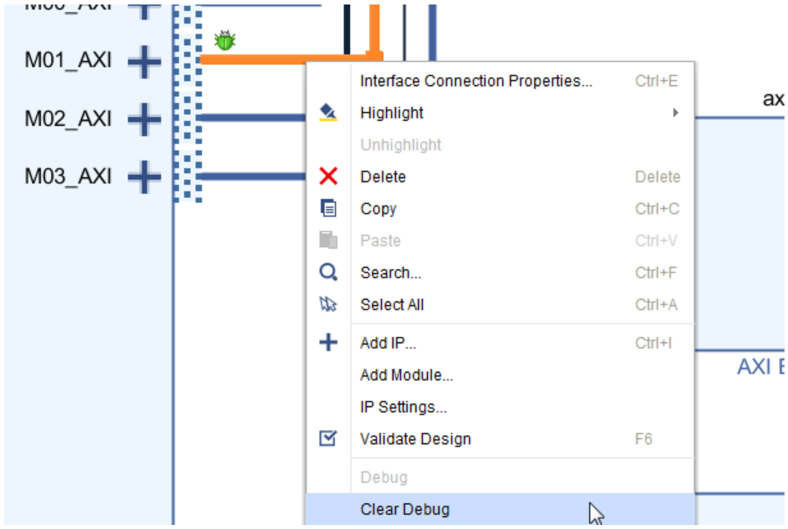
Figure 136: **Setting System ILA Options**

When you click OK on the Run Connection Automation dialog box you see messages such as the following, indicating what action was taken by the tool:

```
Debug Automation : Instantiating new System ILA block '/system_ila_0' with
mode
INTERFACE, 1 slot interface pins and 0 probe pins. Also setting parameters
on this
block, corresponding to newly enabled interface pins and probe pins as
specified via
Debug Automation.
Debug Automation : Connecting source clock pin /clk_wiz_1/clk_out1 to the
following
sink clock pins /system_ila/clk

Debug Automation : Connecting source reset pin
/rst_clk_wiz_1_100M/peripheral_aresetn to the following sink reset pins :
/system_ila_0/resetn

Debug Automation : Connecting interface connection /
microblaze_0_axi_periph_M01_AXI,
to System ILA slot interface pin /system_ila_0/SLOT_0_AXI for debug.
```

After a net has been marked for debug, you can remove the `DEBUG` attribute by right-clicking the net and selecting Clear Debug from the context menu, shown in the following figure. This automatically removes the connection of the selected net to the System ILA, and reconfigures the IP as needed for the appropriate number of Interfaces/Probes.

Figure 137: **Removing Debug Cores from the Block Design**



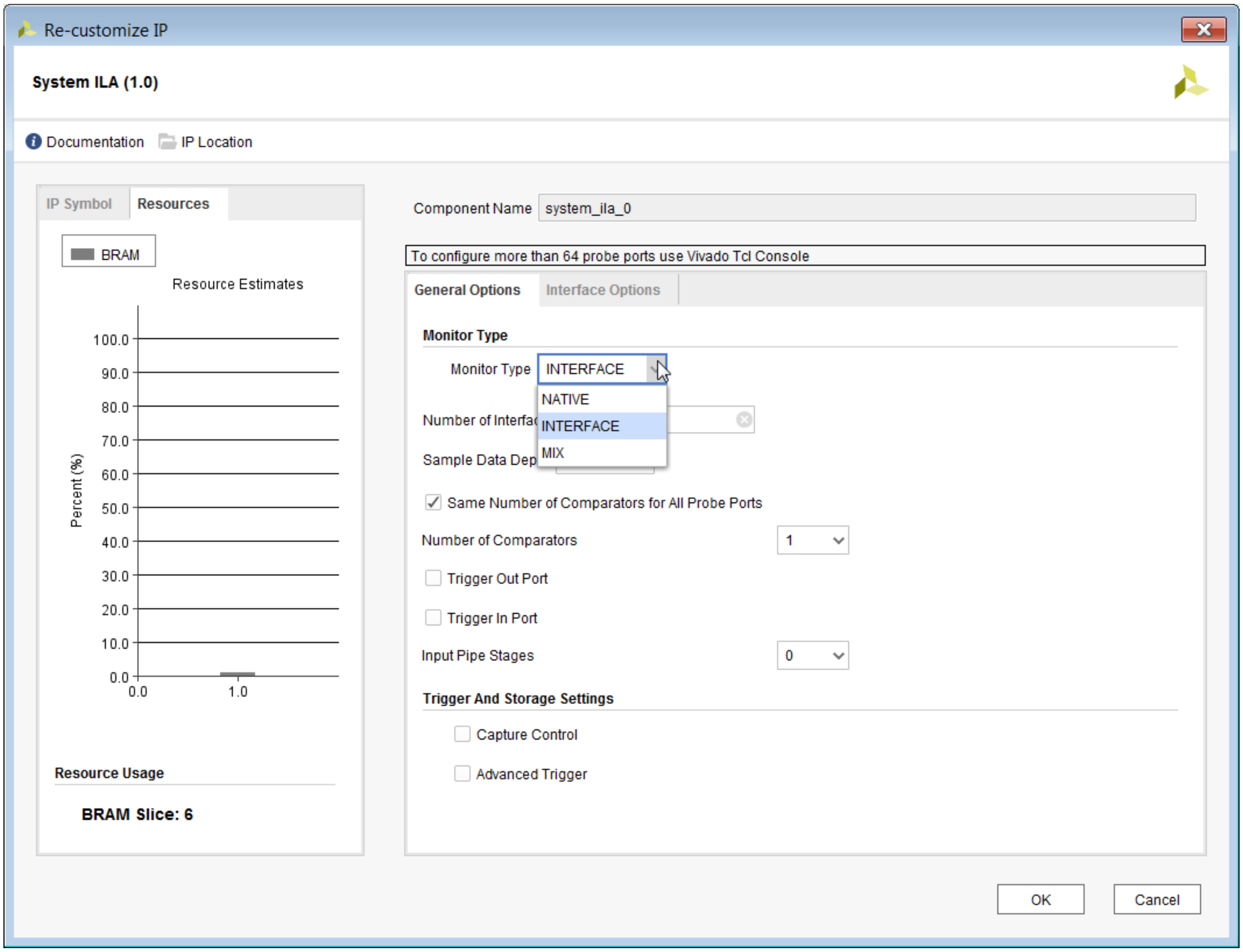
The System ILA IP can also be manually configured to connect nets to debug to the core.

TIP: *While you can manually configure the System ILA IP for the desired number of interfaces/probes and connect the nets to the pins of the ILA, this practice is not recommended.*

Double-click the IP in the block design, or right-click the IP and use the Customize Block command, to re-customize the System ILA IP.

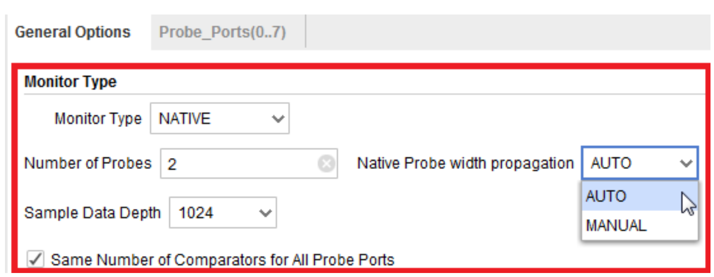
The Re-customize IP dialog box opens for the System ILA debug core as shown in the following figure. The IP Symbol and Resources tab of the System ILA dialog box shows the pins present on the System ILA IP, and the block RAM resources that are consumed by the System ILA debug core.

Figure 138: **The System ILA Configuration Wizard**



The Monitor Type of the IP can be configured as NATIVE for debugging standard signals connected to non-interface pins, INTERFACE for debugging nets connected to interface pins, or MIX for debugging both standard signals and interfaces.

Figure 139: **The System ILA Configuration Wizard**

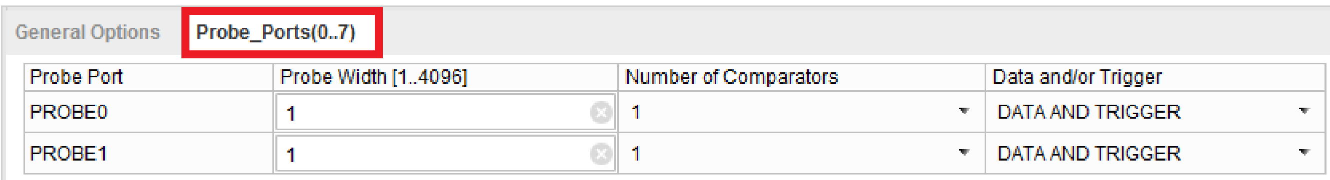


When the Monitor Type selection is NATIVE or MIX, the Number of Probes field is provided to define the number of probes for the debug core, as shown.

These probes can be set to either the AUTO or MANUAL width propagation, which determines how the probe width is determined for a connected signal.

The AUTO mode automatically sets the probe width to the width of the connected signal. When the Native Probe width propagation is set to MANUAL, you must manually set the width of the probes by selecting the Probe Ports tab in the Re-customize IP dialog box and setting the width of the probes, as well as other parameters, as shown below.

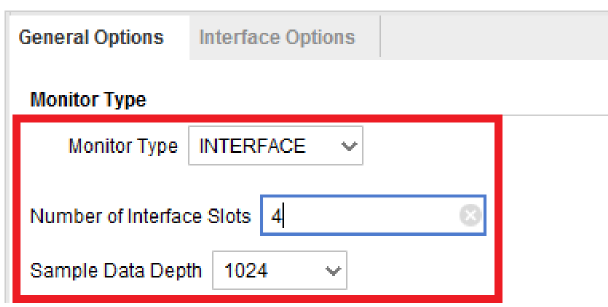
Figure 140: **Setting the System ILA Options in MANUAL Mode Propagation**



When only interface signals are to be debugged by the System ILA, set the Monitor Type field to INTERFACE. When the Monitor Type selection is INTERFACE or MIX, the Number of Interface Slots field displays, which lets you define the number of interface signals to debug.

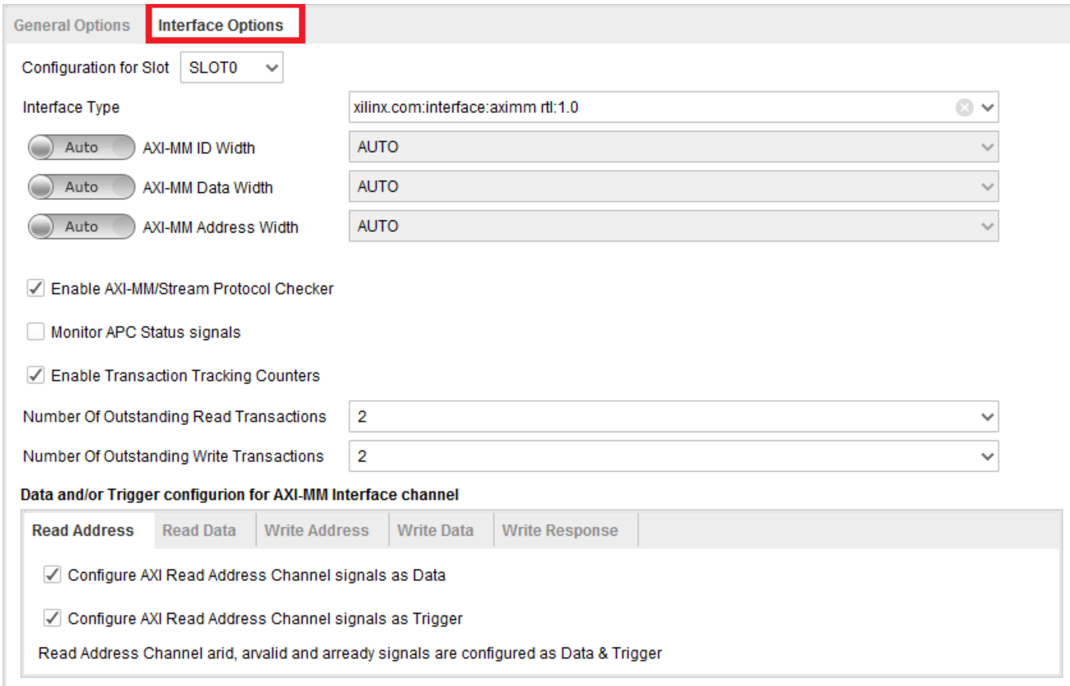
TIP: *The System ILA core can be configured to select up to 1,024 probes, or 16 interface signals, or a mix of probes and interfaces.*

Figure 141: **Setting the System ILA Options When Monitor Type is Set to INTERFACE**



Additionally, the Interface Options tab is added to the Re-customize IP dialog box to let you configure the interface slots as shown in the following figure. You can also set other parameters for debugging interfaces from the Interface Options tab. The options displayed can change based on the type of interface being debugged.

Figure 142: **Setting the System ILA Options using the Interface Options Tab**



When the Monitor Type field is set to MIX, both the Probe Options and the Interface Options tabs display, as shown.

After the nets have been marked and connected to the System ILA IP, you will need to validate the design. Validating the design ensures that all debug nets and their associated clocks are correctly connected to the System ILA.

The Validate Design command returns the following warning message:

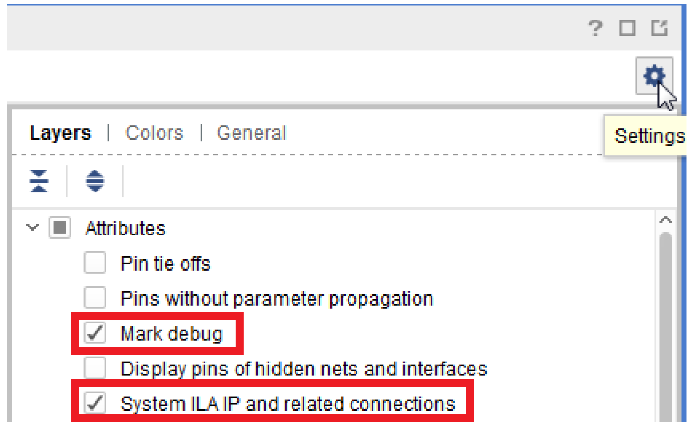
```
WARNING: [BD 41-1781] Updates have been made to one or more nets/interface
connections marked for debug. Debug nets, which are already connected to
System ILA
IP core in the block-design, will be automatically available for debug in
Hardware
Manager. For unconnected Debug nets, please open synthesized design and use
'Set Up
Debug' wizard to insert, modify or delete Debug Cores. Failure to do so
could result
in critical warnings and errors in the implementation flow.
```

This warning message can be safely ignored if you used Designer Assistance to connect all nets marked for debug to one or more System ILA cores. Any errors returned by Validate Design should be examined and resolved.

If you have marked nets for debug that are not connected to a System ILA, use the Netlist Insertion flow to connect those signals to an ILA debug core in the top-level design. See Using the Netlist Insertion Flow for more information.

You can easily see which nets are marked for debug, and which nets are connected to the System ILA debug core by using the Layers view to display the nets, as shown in the following figure. See Displaying Layers in the Block Design for more information.

Figure 143: **Viewing Nets Marked for Debug and System ILA Connectivity using Layers View**



After the block design is successfully validated, you can create the HDL wrapper, and take the top-level design through synthesis and implementation. See Integrating the Block Design into a Top-Level Design.

TIP: *Additional details of debugging AXI interfaces in the Vivado Hardware Manager are described at this link in the Vivado Design Suite User Guide: Programming and Debugging (UG908)*

Using the ILA IP to Debug a Block Design

IMPORTANT! *Existing block designs can continue to use the Integrated Logic Analyzer (ILA) debug core. However, new block designs should use the System ILA debug core as described at Using the System ILA IP to Debug a Block Design.*

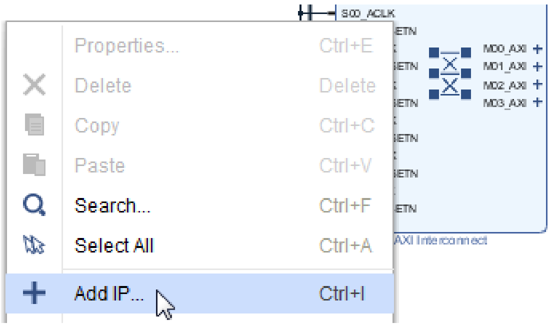
If an ILA debug core is found in the block design, you will see the following INFO message:

```
[xilinx.com:ip:ila:6.2 6] /ila_0: Xilinx recommends using the System ILA IP
in IP
Integrator. The System ILA IP is functionally equivalent to an ILA and
offers
additional benefits in debugging interfaces both within IP Integrator and
the
Hardware Manager. Consult the Programming and Debug User Guide UG908 for
further
details.
```

You can instantiate an Integrated Logic Analyzer (ILA) in the IP integrator design, and connect nets that you are interested in probing to the ILA.

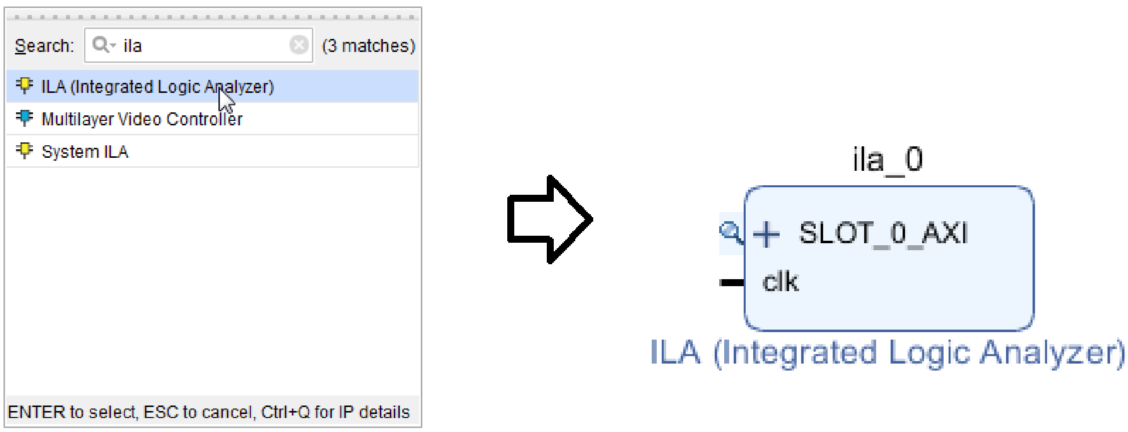
Use the following steps to instantiate an ILA:

1. Right-click the block design canvas and select Add IP, as shown in the following figure.



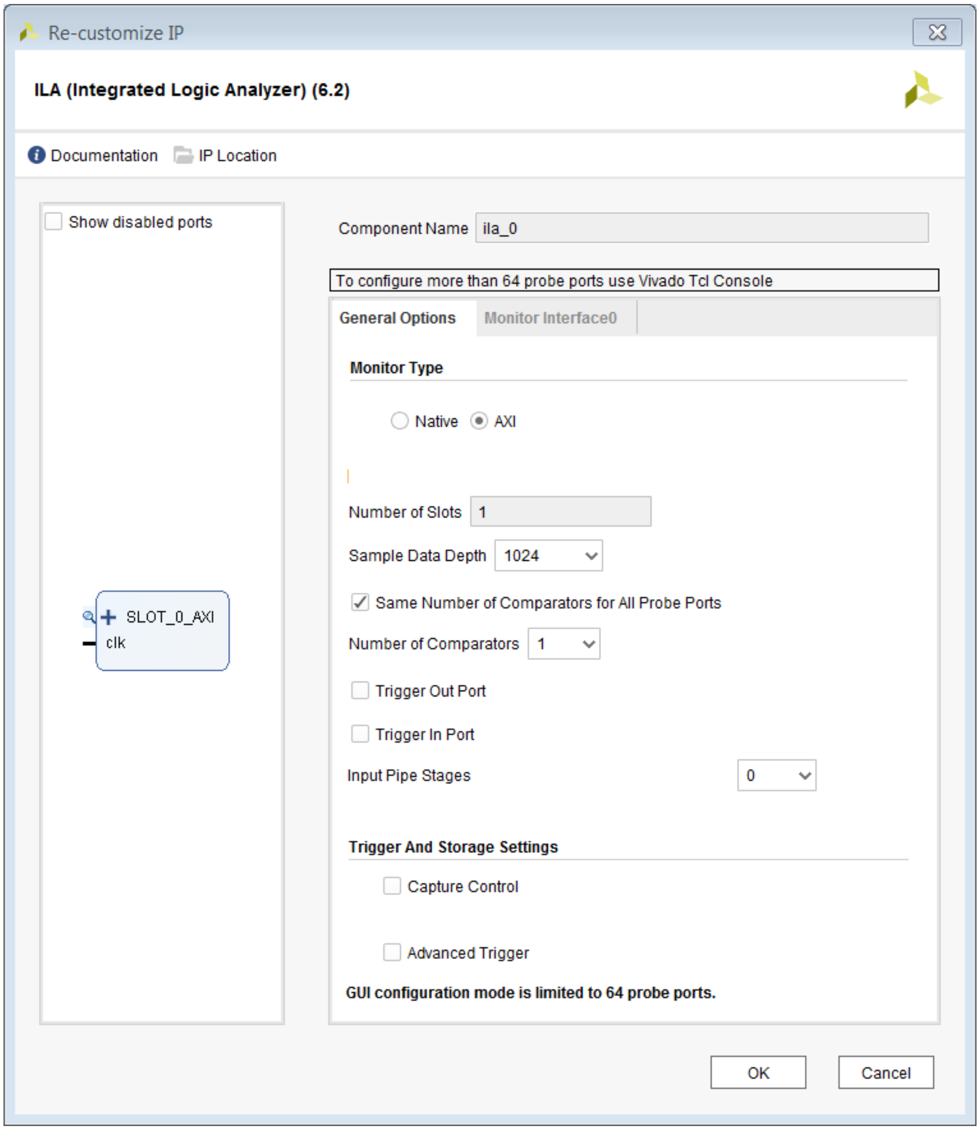
2. In the IP catalog, type ILA in the search field, select and double-click the ILA core to instantiate it on the IP integrator canvas.

   The following figure shows the ILA core instantiated on the IP integrator canvas.



3. Double-click the ILA core to reconfigure it.

   The Re-Customize IP dialog box opens, as shown in the following figure.



The default option under the General Options tab shows AXI as the Monitor Type.

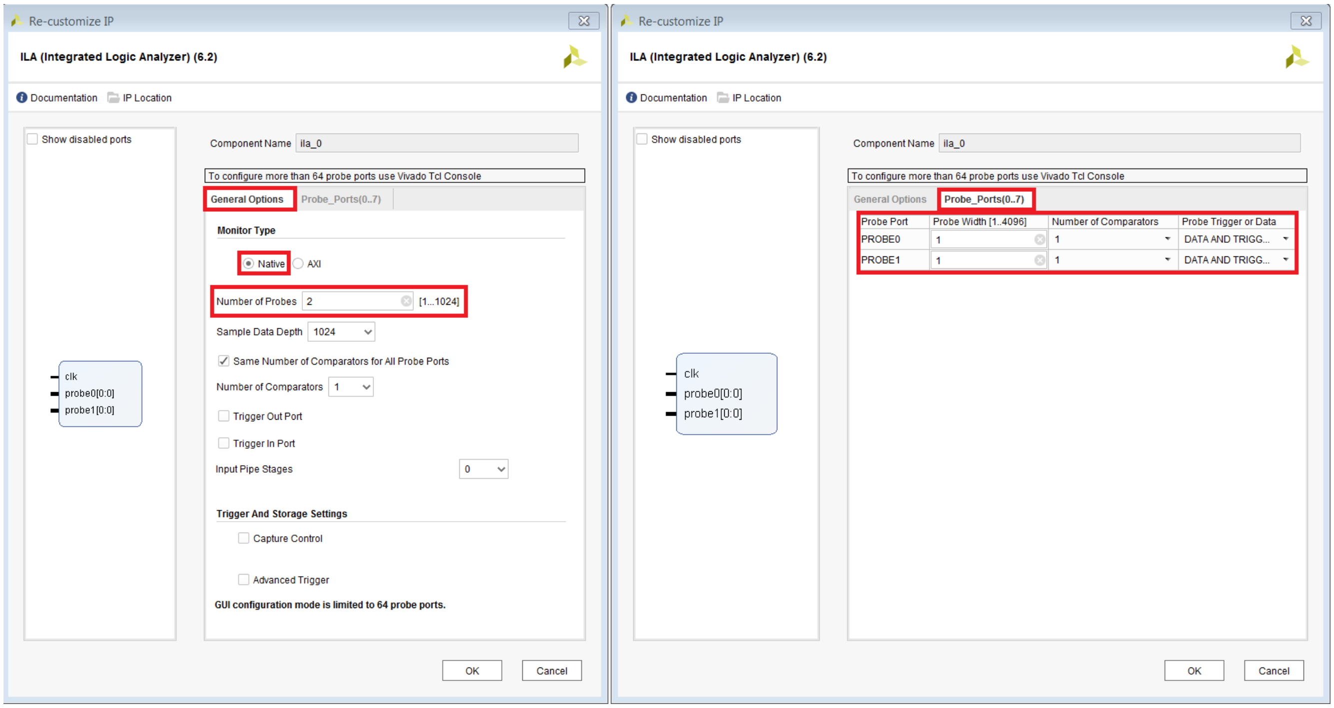
- If you are monitoring an entire AXI interface, keep the Monitor Type as AXI.
- If you are monitoring non-AXI interface signals, change the Monitor Type to Native.

You can change the Sample Data Depth and other fields as desired. For more information, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

CAUTION! *You can only monitor one AXI interface using an ILA. Do not change the value of the Number of Slots. If you need to debug more than one AXI interface, then instantiate more ILA cores as needed.*

When you set the Monitor Type to Native, you can set the Number of Probes value, as shown in the following figure. Set this value to the number of signals you want to be monitored.

Figure 144: **ILA Native Monitor Type Option**

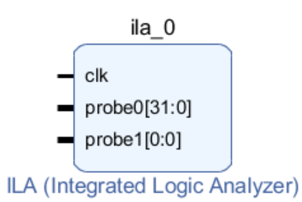


The Number of Probes is set to 2 in the General Options tab. You can see under the Probe_Ports tab that two ports display. The width of these ports can be set to the desired value.

4. Assuming that you want to monitor a 32-bit bus, set the Probe Width for `Probe0` to 32.

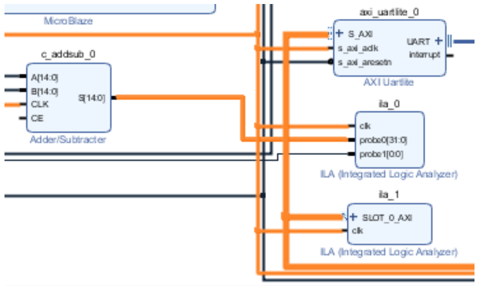
   After you configure the ILA, the changes are reflected on the IP integrator canvas as shown in the following figure.

Figure 145: **ILA Core after Changes in the Re-customize IP Dialog Box**



5. After configuring the ILA, make the required connections to the pins of the ILA on the IP integrator canvas, as shown.

Figure 146: **Instantiated ILAs to Monitor AXI and Non-AXI Signals**



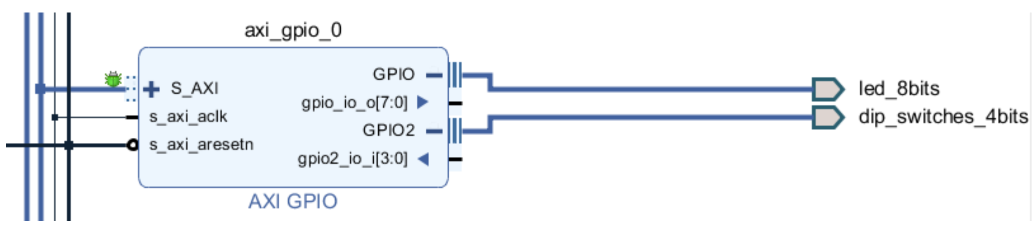
> **CAUTION!** *If a pin connected to an I/O port is to be debugged, use* `MARK_DEBUG` *to mark the nets for debug. The following section describes this action.*

6. Follow on to synthesize, implement, and generate bitstream.

Often, the I/O ports of a block design need to be probed for debugging. If the I/O ports of interest are bundled into interface ports then you must take care when connecting these interface ports or pins to the ILA or VIO debug core. You must pull the signals of interest out of the bundled interface port or pin. For more information, see Connecting Interface Signals.

As an example, consider the MicroBlaze processor design for the KC705 board, shown in the following figure. This design has a GPIO configured to use both the 8-bit LED interface and the 4-bit dip switches on the KC705 board.

Figure 147: **Monitoring Interface Signals in a Block Design**



To monitor these I/O interfaces, do the following:

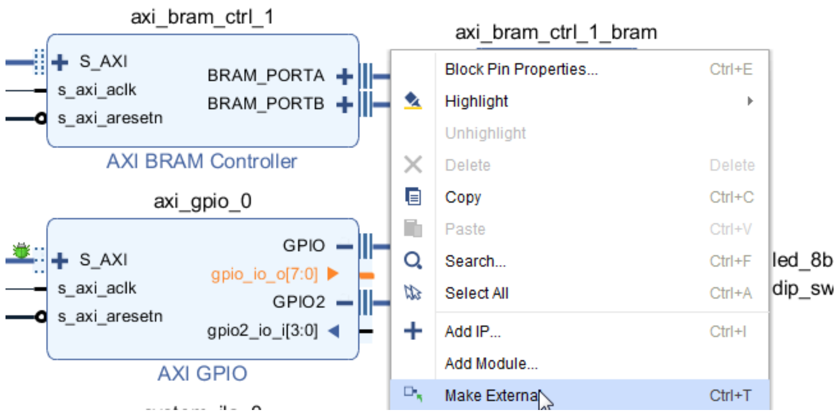
1. Expand the GPIO interface pins so that you can see the individual signals that make up the interface pin.

   As you can see in the following figure, the GPIO interface consists of an 8-bit output pin (`gpio_io_o[7:0]`), and the GPIO2 interface consists of a 4-bit input pin (`gpio2_io_i[3:0]`).

   To monitor these pins using debug probes you need to make them external to the block design. In other words, you must tie the pins inside the interface pin to an external port.

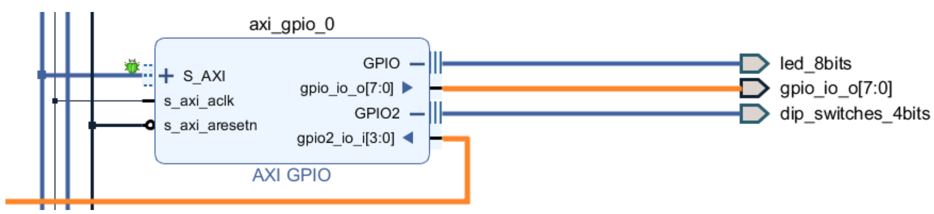
2. Right-click the pin, and select **Make External**.

Figure 148: **Using Make External Command to Connect I/O Pin to an I/O Port**



You can see in the following figure that the pins that make up the `GPIO` and `GPIO2` interface pins have been tied to external ports in the block design. Next, you must connect these pins to an ILA debug core.

Figure 149: **External Ports Connected to Pins**



CAUTION! *When you make the I/O pins of an interface external, by connecting the input or output pins to external ports, do not delete the connection between the top-level interface pin and the I/O port. As shown in the following figure, leave the existing top-level interface pin connected externally to the appropriate interface.*

When connecting to individual signals or buses of an interface, you will see a warning as shown below:

```
WARNING: [BD 41-1306] The connection to interface pin /axi_gpio_0/
gpio2_io_i is being
overridden by the user. This pin will not be connected as a part of
interface
connection GPIO2.
```

You must manually connect all of the pins of this individual signal or bus, as they will no longer be connected as part of the bundled interface.

IMPORTANT! *This is an especially important concept when adding an ILA or VIO core to probe a signal. Often you will simply connect the ILA or VIO core to one pin of an interface, without realizing you have removed that signal from the bundled interface. The signal connection is broken unless you connect to other expanded interface pins as needed.*

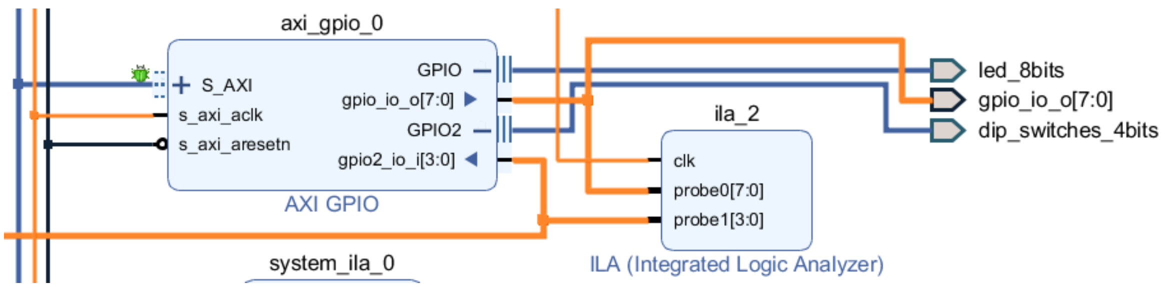
3. Use the Add IP command to instantiate an ILA core into the design, and configure it to support either Native or AXI mode.

   ***Note*:** In this case you must configure the ILA to support Native mode because you are not monitoring an AXI interface.

4. Configure two probes on the ILA core:
   - One that is 8-bits wide to monitor the LED
   - One that is 4-bits wide to monitor the DIP Switches

5. Connect the ILA probes to the appropriate input/output pins, and connect the ILA clock to the same clock domain as that of the I/O pins, as shown in the following figure.

Figure 150: **ILA Probes Connected to the Input/Output Pins for Monitoring**



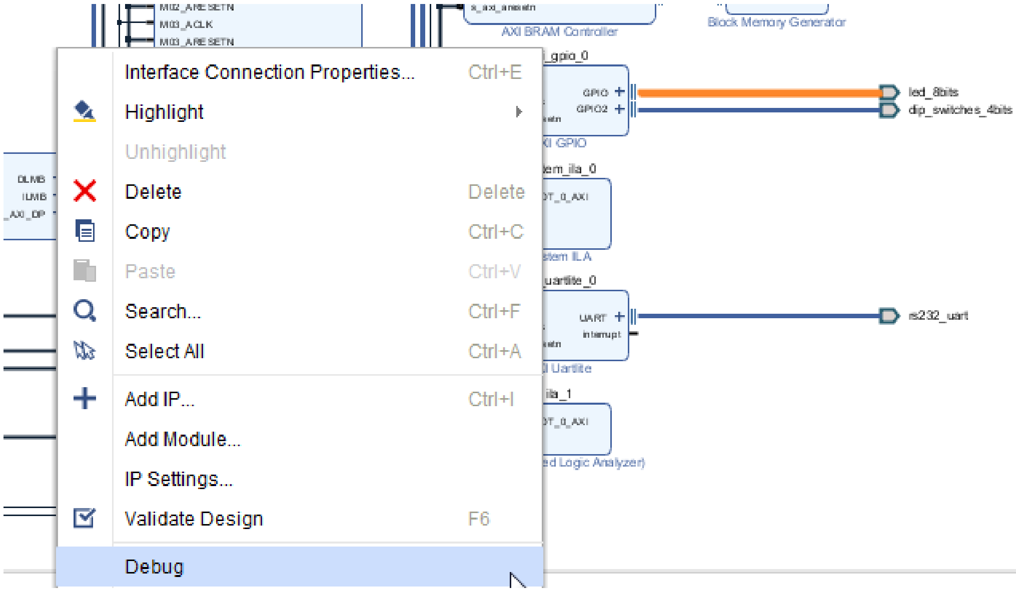
With the debug cores inserted into the block design, the generated output products will include the necessary logic and signal probes to debug the design in the Vivado hardware manager. For more information on working with the Vivado hardware manager, and programming and debugging devices, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Using the Netlist Insertion Flow

In this flow, you mark the nets in the block design for debug that you are interested in analyzing in the Vivado Hardware Manager. Marking nets for debug in the block design offers more control in terms of identifying debug signals during coding, and enabling/disabling debugging after the netlist has been generated.

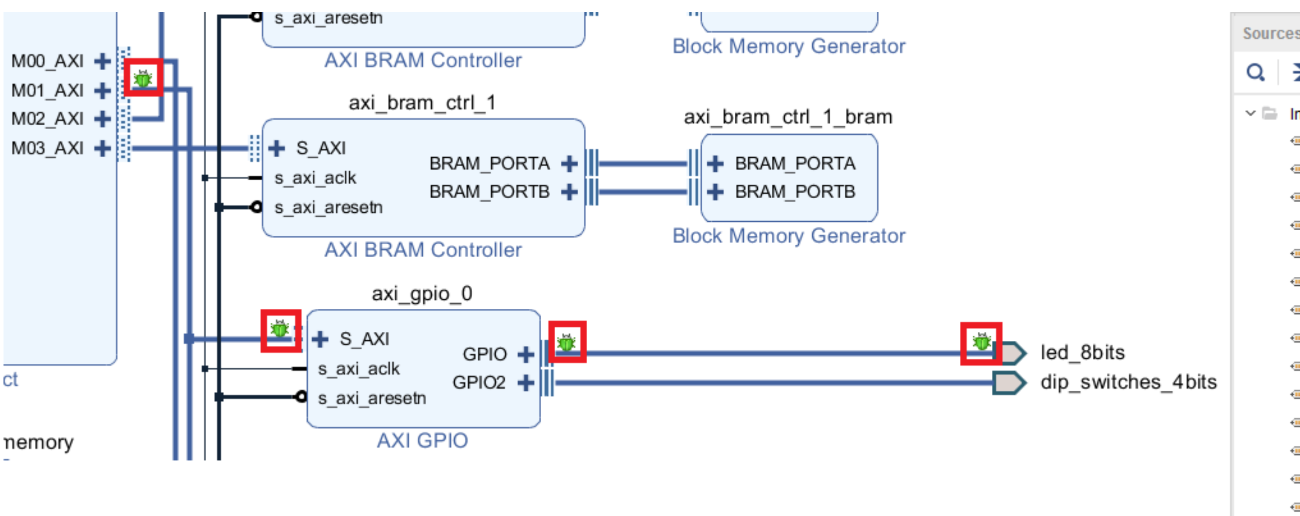
Marking Nets for Debug in the Block Design

1. To mark nets for debug, in the block design:
   a. Highlight the net.
   b. Right-click and select **Debug**, as shown in the following figure.



The nets that are marked for debug show a small bug icon placed on top of the net in the block design.

Also, a bug icon is placed on the nets to be debugged in the Design window, as shown in the following figure.



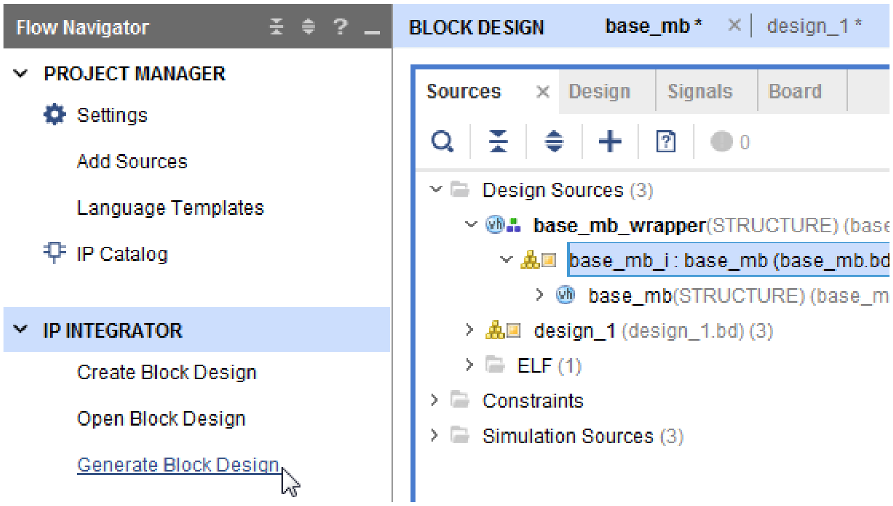
2. To mark multiple nets for debug at the same time:
    a. Highlight the nets together.
    b. Right-click, and select **Mark Debug**.

Generating Output Products

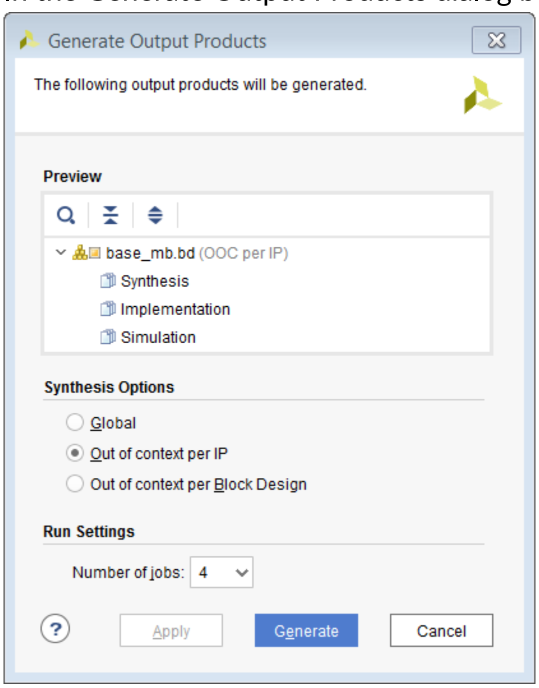
You can Generate Output Products as follows:

1. In the Flow Navigator, click **Generate Block Design**.

   Alternatively, you can highlight the block design in the sources window, right-click and select **Generate Output Products**, as shown in the following figure.



2. In the Generate Output Products dialog box, shown in the following figure, click **Generate**.



   When you mark the nets for debug, the DEBUG and MARK_DEBUG attributes are placed on the net, which can be seen in the generated top-level HDL file, shown in the following figure. This prevents the Vivado tools from optimizing and renaming the nets.

```
attribute DEBUG : string;
attribute DEBUG of axi_gpio_0_GPIO_TRI_O : signal is "true";
attribute MARK_DEBUG : boolean;
attribute MARK_DEBUG of axi_gpio_0_GPIO_TRI_O : signal is std.standard.true;
```

Synthesize the Design and Insert the ILA Core

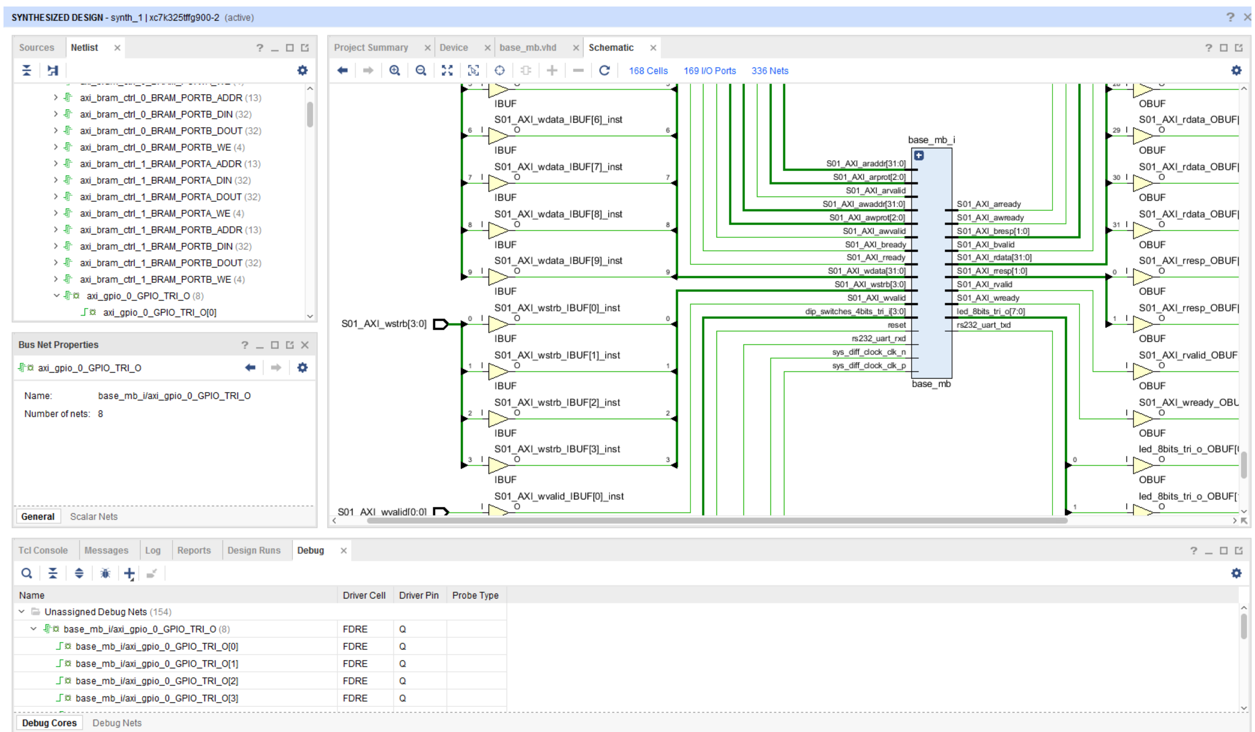
The next step is to synthesize the top-level design. To do this:

1. Select **Flow Navigator → Synthesis**, and click **Run Synthesis**.

   After synthesis finishes, the Synthesis Completed dialog box opens.

2. Select **Open Synthesized Design** to open the netlist design, and click **OK**.

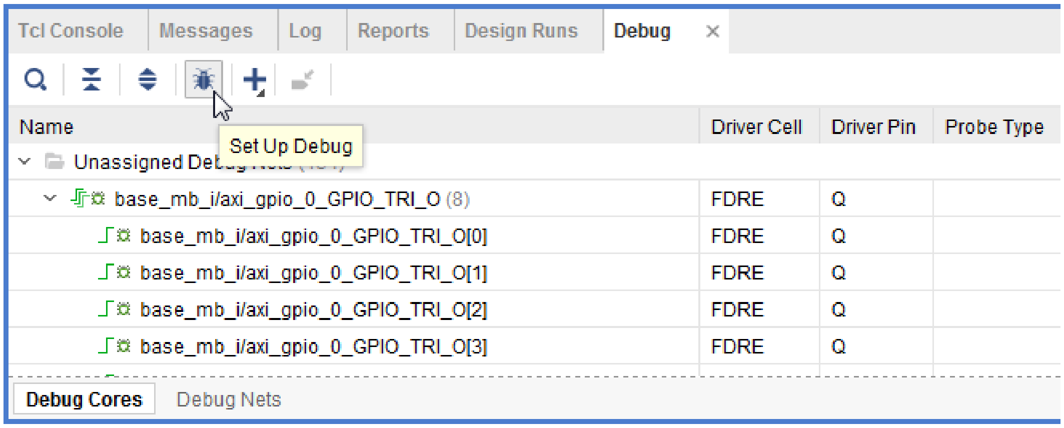
   The Schematic and the Debug window opens. If the Debug window at the bottom of the GUI is not open, you can always open that window by choosing Windows > Debug from the menu. The following figure shows the Debug window.



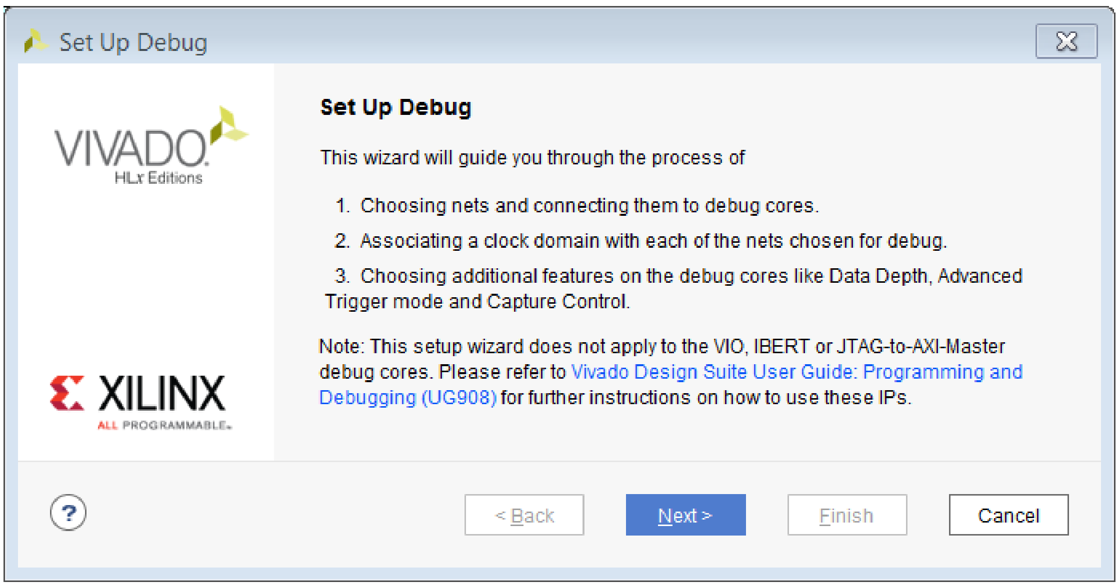
   You can see all the nets that were marked for debug in the Debug window under the folder Unassigned Debug Nets. These nets need to be connected to the probes of an Integrated Logic Analyzer (ILA). This is the step where you insert an ILA core and connect these unassigned nets to the probes of the ILA.

3. Click the Set up Debug button  in the Debug window toolbar.

   Alternatively, you can also select **Tools → Set Up Debug**, shown in the following figure.

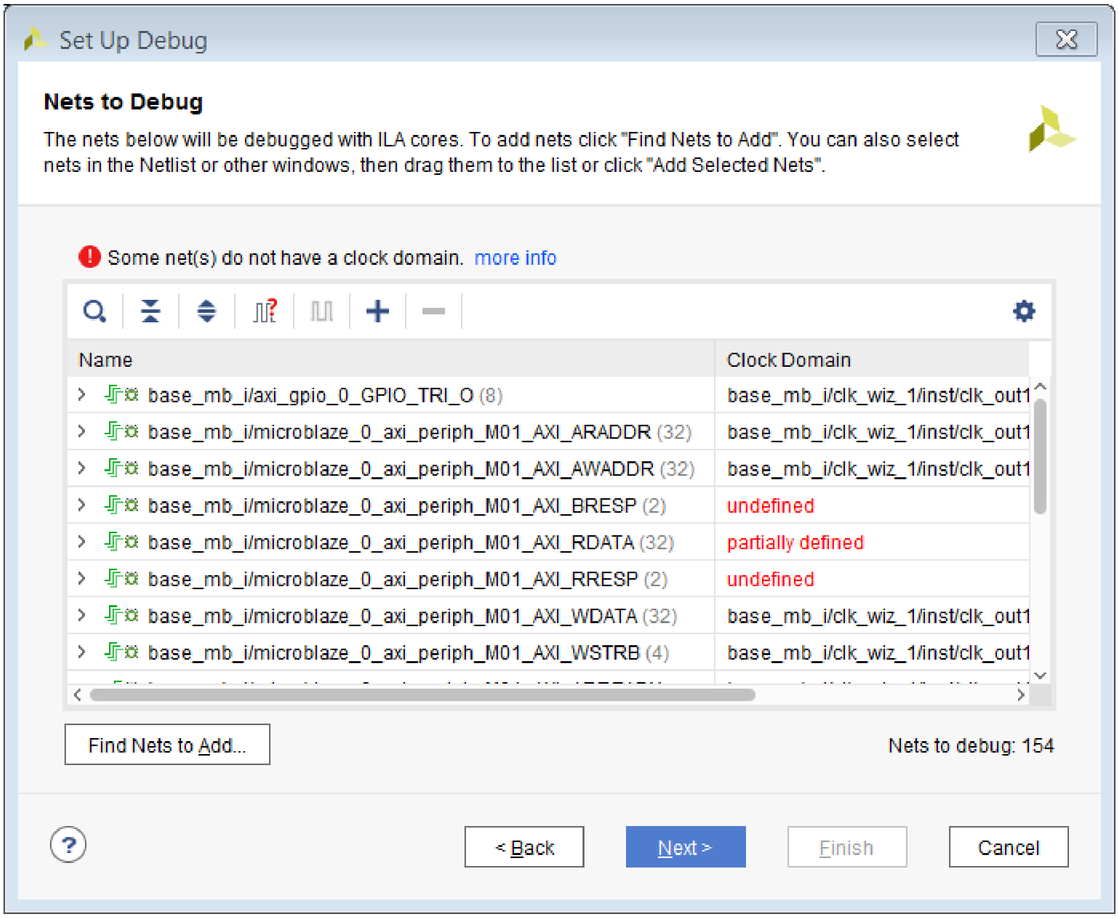


The Set Up Debug wizard opens, as shown in the following figure.



4. Click **Next**.

   The Nets to Debug page opens, as shown in the following figure.

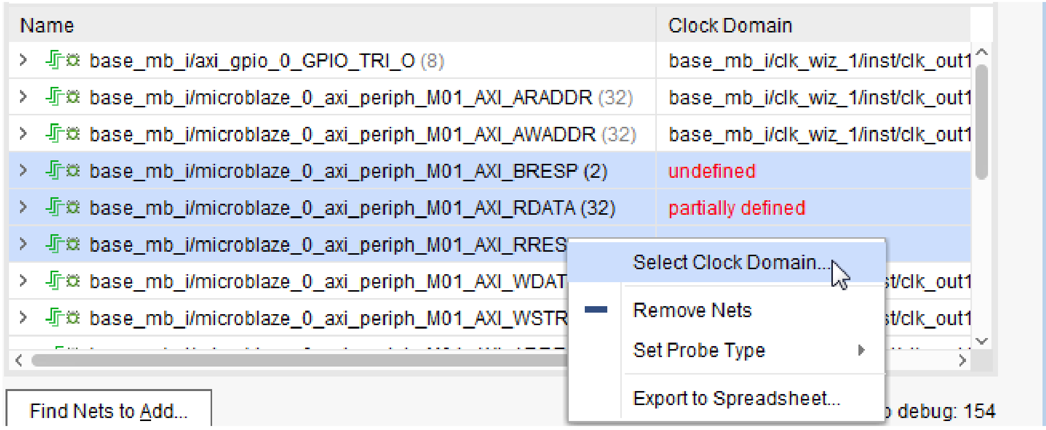


5. Select a subset (or all) of the nets to debug. Every signal must be associated with the same clock in an ILA. If the clock domain association cannot be found by the tool, manually associate those nets to a clock domain by selecting all the nets that have the Clock Domain column specified as undefined or partially defined.

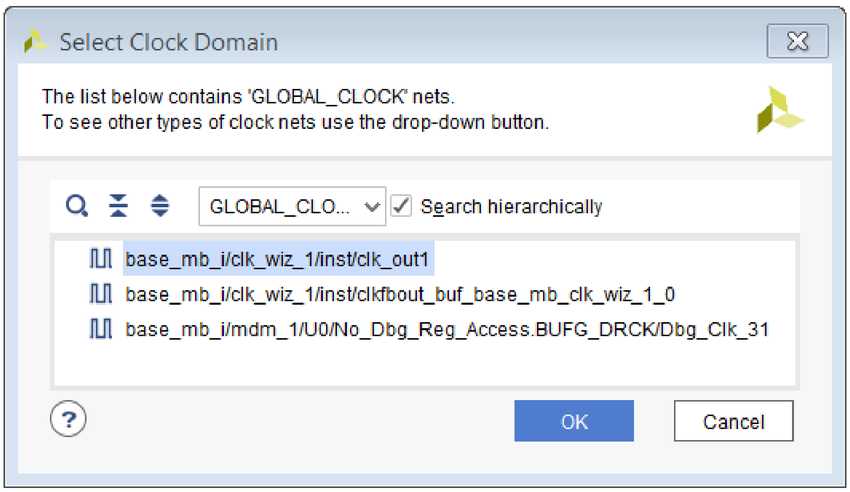
CAUTION! *You need to mark the entire interfaces that you are interested in debugging; however, if you are concerned with device resource usage, then the nets you do not need for debugging can be deleted while setting up the debug core.*

6. To associate a clock domain to the signals that have an undefined or partially defined Clock Domain, select the nets, right-click, and choose **Select Clock Domain** as shown in the following figure.

TIP: *One ILA is inferred per clock domain by the Set up Debug wizard.*

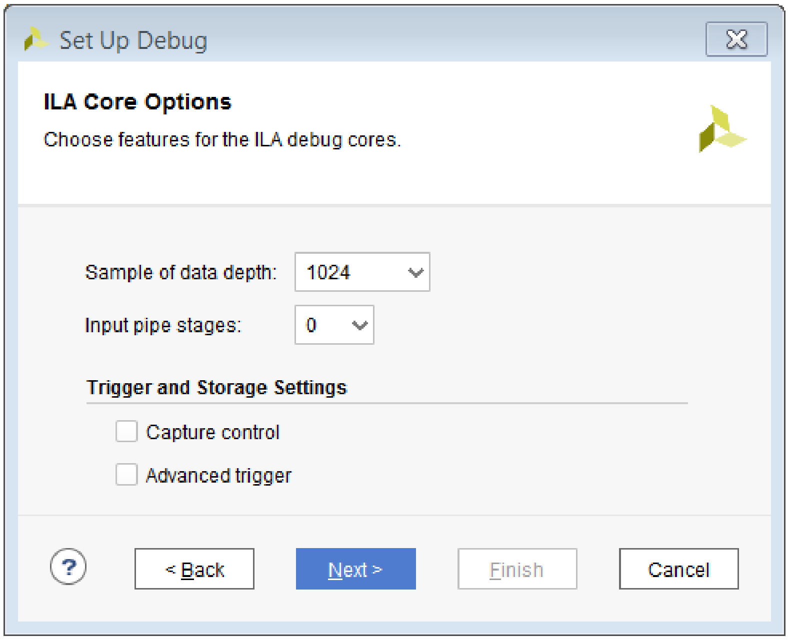


7. In the Select Clock Domain dialog box, shown in the following figure, select the clock, and click **OK**.



8. In the Specify Nets to Debug dialog box, click **Next**.

9. In the ILA Core Options page, shown in the following figure, select the appropriate options for triggering and capturing data, and click **Next**.

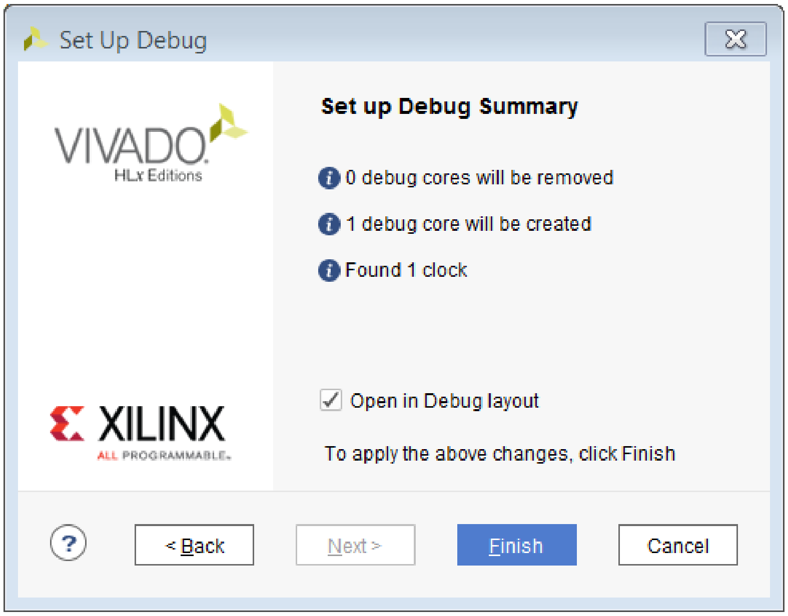


The advanced triggering capabilities provide additional control over the triggering mechanism. Enabling advanced trigger mode enables a complete trigger state machine language that is configurable at run time.

There is a three-way branching per state and there are 16 states available as part of the state machine. Four counters and four programmable counters are available and viewable in the Analyzer as part of the advanced triggering.

In addition to the basic capture of data, capture control capabilities let you capture the data at the conditions where it matters. This ensures that unnecessary block RAM space is not wasted and provides a highly efficient solution.

10. In the Summary page, shown in the following figure, verify that all the information looks correct, and click **Finish**.

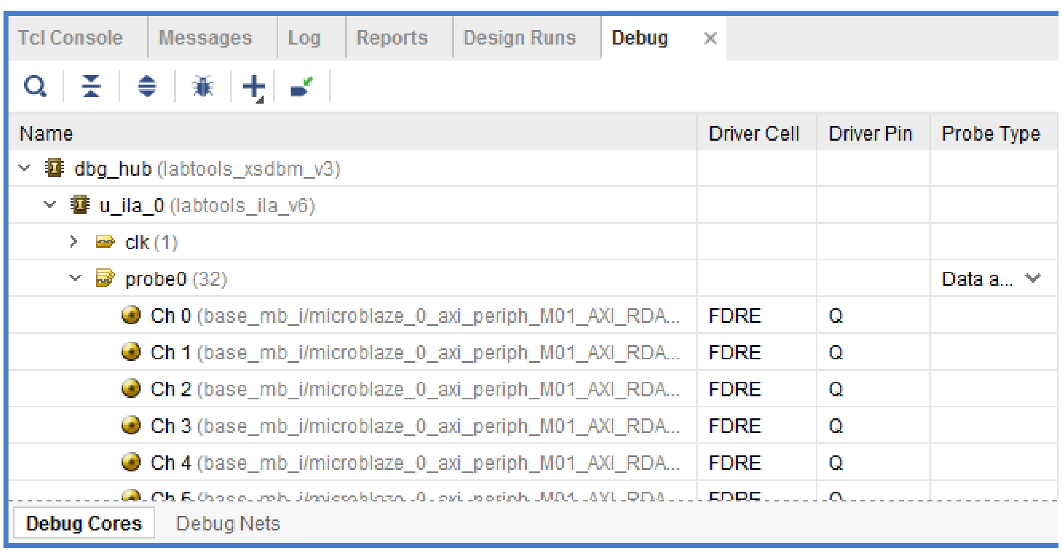


The Debug window looks like the following figure after the ILA core has been inserted.

***Note*:** All the buses (and single-bit nets) have been assigned to different probes.

The probe information also shows how many signals are assigned to that particular probe.

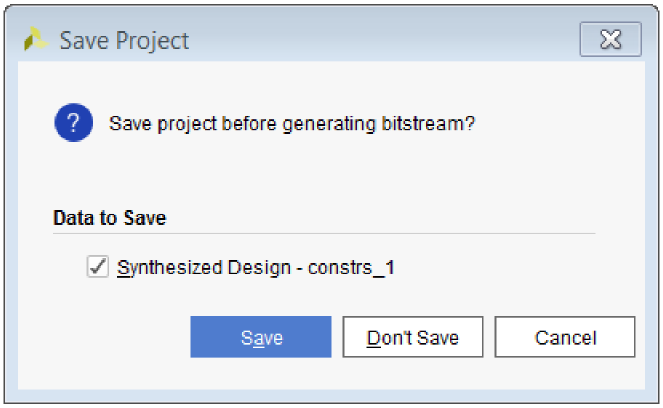
For example, in the following figure, `probe0` has 32 signals (the 32 bits of the `microblaze_1_axi_periph_m02_axi_WDATA`) assigned.



You are now ready to implement your design and generate a bitstream.

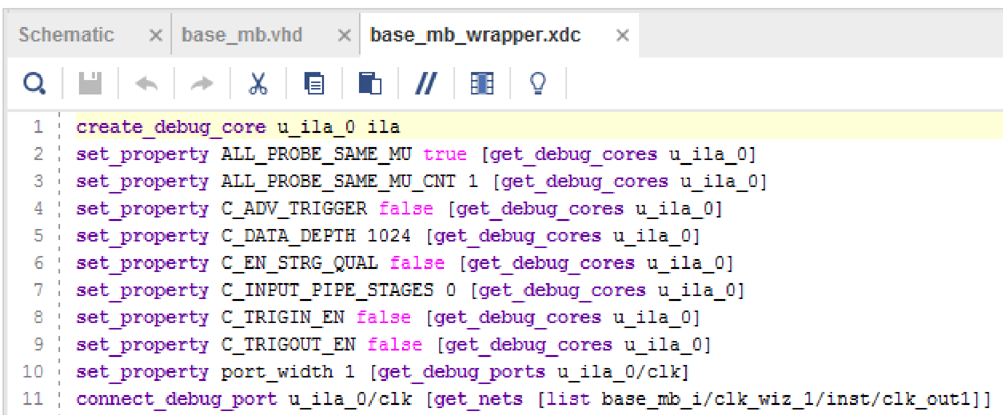
11. Select **Flow Navigator → Program and Debug**, and click **Generate Bitstream**.

    Because you made changes to the netlist (by inserting an ILA core), a dialog box, as shown in the following figure, displays asking if the design should be saved prior to generating bitstream.



You can choose to save the design at this point, which writes the appropriate constraints in an active constraints file (if one exists), or create a new constraints file.

The constraints file contains all the commands to insert the ILA core in the synthesized netlist as shown in the following figure.



The benefit of saving the project is that if the signals marked for debug remain the same in the original block design, then there is no need to insert the ILA core after synthesis manually as these constraints will take care of it. Therefore, subsequent iteration of design changes will not require a manual core insertion.

If you add more nets for debug (or unmark some nets from debug) then you must open the synthesized netlist and make appropriate changes using the Set up Debug wizard.

If you do not chose to save the project after core insertion, none of the constraints show up in the constraints file and you must insert the ILA core manually in the synthesized netlist in subsequent iterations of the design.

With the debug cores and signal probes inserted into the top-level design, you are ready to debug the design in the Vivado hardware manager. For more information on working with the Vivado hardware manager, and programming and debugging devices, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Removing Debug Logic after Debug

You can remove debug logic in several different ways, depending on the chosen flow:

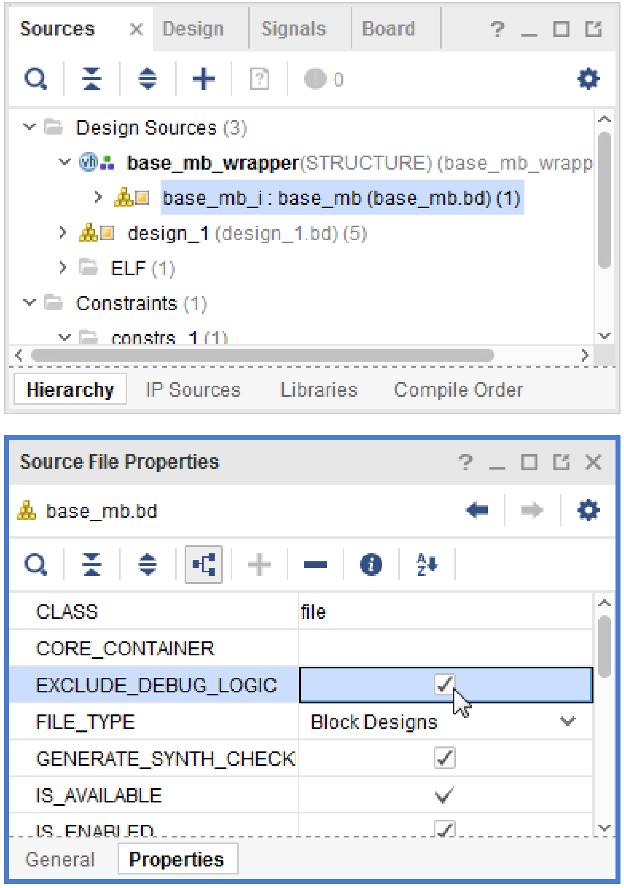
- If HDL instantiation was done with System ILA, then select and right-click the net marked for debug in the block design.
- Clear Debug option can be selected from the context menu. This removes the connection between the net marked for debug and the System ILA and also re-configures the ILA to debug only the other nets. If there are no nets to be debugged, then the System ILA is deleted.

In some cases, you might want to keep the debugging logic within the block design as it is, but, want to exclude the debugging logic from the generated HDL. To support this, block designs have an `EXCLUDE_DEBUG_LOGIC` property, which can be enabled in the Properties window or through the `set_property` Tcl command, specified as follows:

```
set_property EXCLUDE_DEBUG_LOGIC 1 [get_files
C:/Temp/base_mb_kc705/base_mb_kc705.srcs/sources_1/bd/base_mb/base_mb.bd]
```

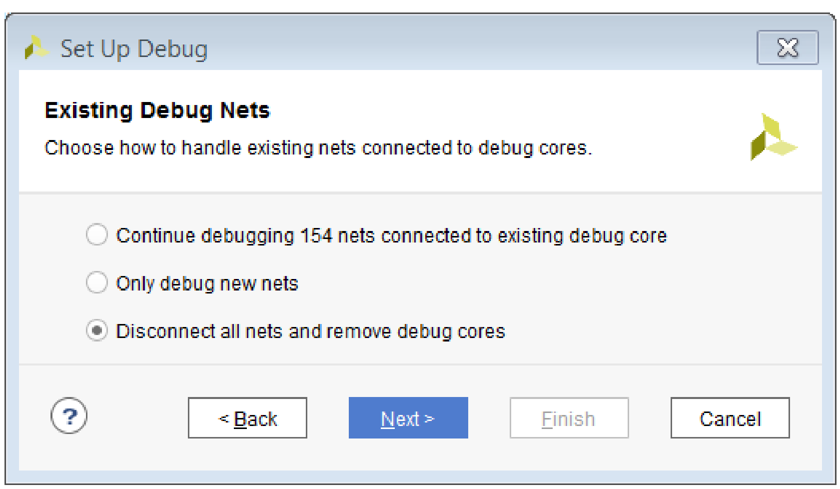
With the block design selected in the Sources window, check the `EXCLUDE_DEBUG_LOGIC` property in the Source File Properties window, as shown in the following figure.

Figure 151: **Excluding Debug Logic from Generation**



If netlist insertion flow was used to insert an ILA after synthesis, then you must remove the ILA manually. To do this, open the netlist after synthesis and in the Existing Debug Nets page of the Debug wizard, select Disconnect all nets and remove debug cores.

Figure 152: **Removing Debug Cores in the Insertion Flow**



Chapter 8

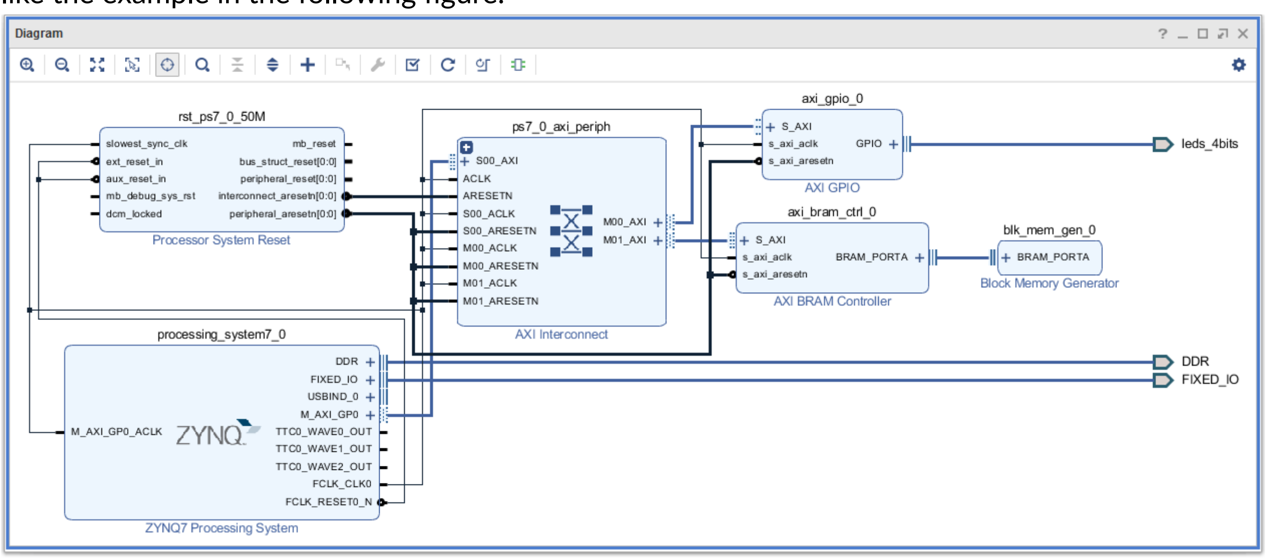
Using Tcl Scripts to Create Projects and Block Designs

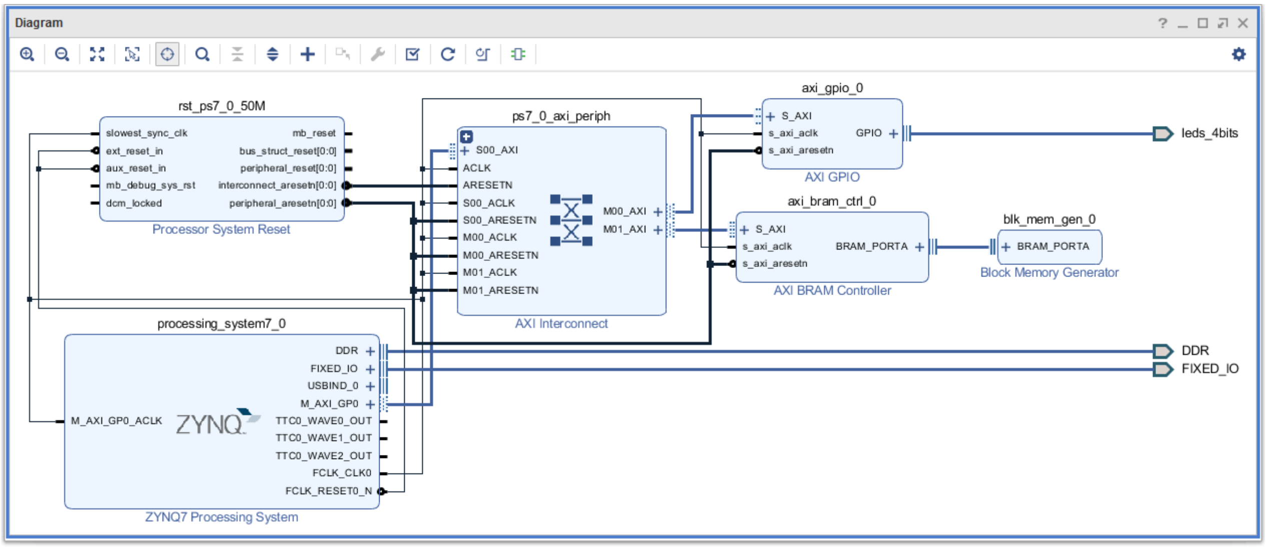
Typically, you create a new design in a project-based flow in the Vivado® Integrated Design Environment (IDE). After you assemble the initial design, you might want to re-create the design using a scripted flow in the GUI or in batch mode. This chapter guides you through creating a scripted flow for block designs.

Exporting a Block Design to a Tcl Script in the IDE

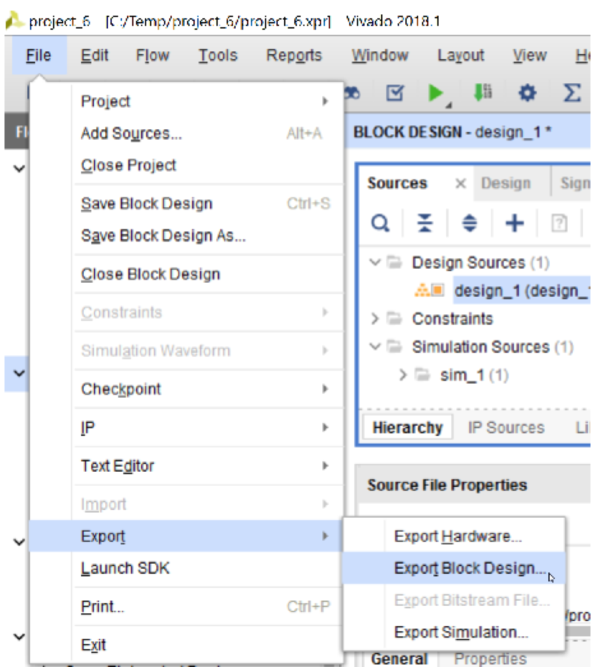
To convert a block design to a Tcl script in the IDE, to the following:

1. Create a project and a new block design in the Vivado IDE as described in Chapter 2: Creating a Block Design. When the block design is complete, your canvas contains a design like the example in the following figure.

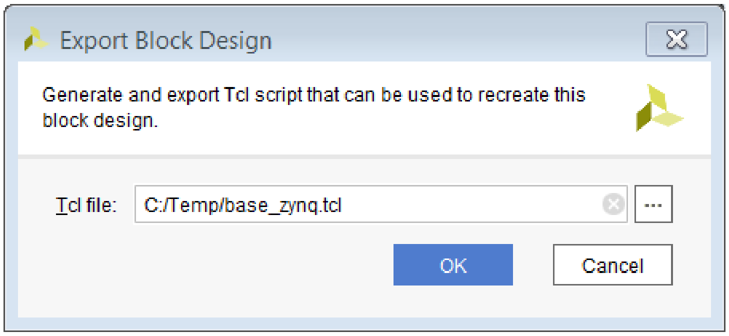




2. With the block design open, select File > Export > Export Block Design, as shown in the following figure.



3. Specify the name and location of the Tcl file in the Export Block Design dialog box, shown in the following figure.



Alternatively, you can type the write_bd_tcl command in the Tcl Console:

```
write_bd_tcl <path to file/filename>
```

This creates a Tcl file that can be sourced to re-create the block design.

***Note*:** Only parameters changed by the user are written out in this Tcl file. The default parameters of a IP are not written out.

Block Design layout information is not written out by default. Instead, you can use an optional `-include_layout` switch with the Tcl command to write out the layout information of blocks within a block design.

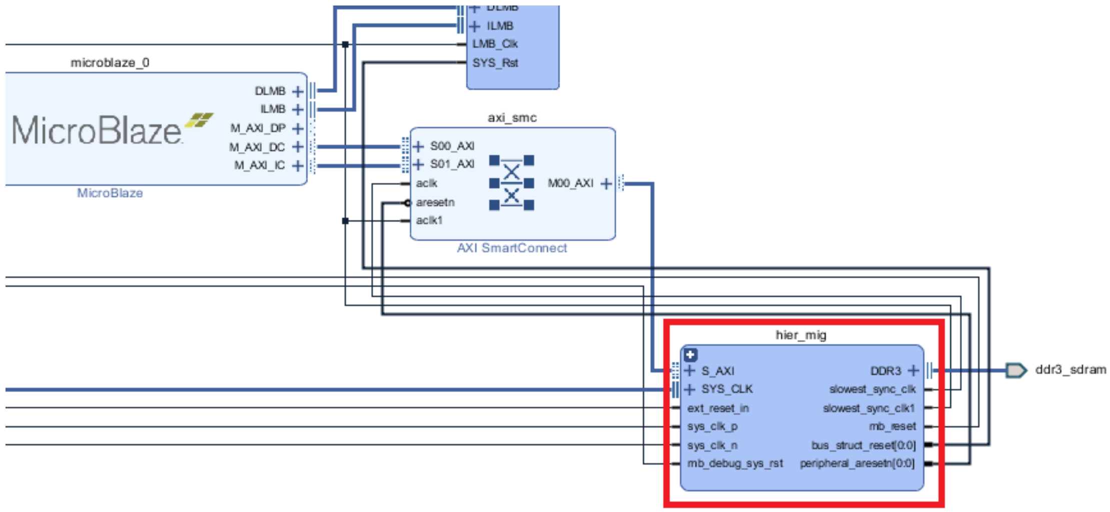
```
write_bd_tcl -include_layout <path to file/filename>
```

This Tcl file has embedded information about the version of the Vivado tools in which it was created, and, consequently this design cannot be used across different releases of the Vivado Design Suite. The Tcl file also contains information about the IP present in the block design, their configuration, and the connectivity.

CAUTION! *Use the script produced by `write_bd_tcl` in the release in which it was created only. The script is not intended for use in other versions of the Vivado Design Suite.*

The `write_bd_tcl` command also provides with the ability to write out Tcl scripts for hierarchical blocks only. This could be useful in situations where a sub-block or hierarchy of a design needs to be reused in some other block design. As an example, looking at the following figure, you want to write out the Tcl script for generating the contents of the hierarchical block, `hier_mig`.

Figure 153: **Writing Out Tcl for a Hierarchy**



This could be done by using the `-hier_blk` switch with the write_bd_tcl Tcl command. For example:

```
write_bd_tcl -hier_blks [get_bd_cells /hier_mig] ./mig_hierarchy.tcl
```

The Tcl script generated from the command above can then be sourced in another block design to create the same hierarchy. In the Tcl Console, type:

```
source ./mig_hierarchy.tcl
```

When this Tcl procedure executes you see the following at the end of the Tcl procedure (in the Tcl console):

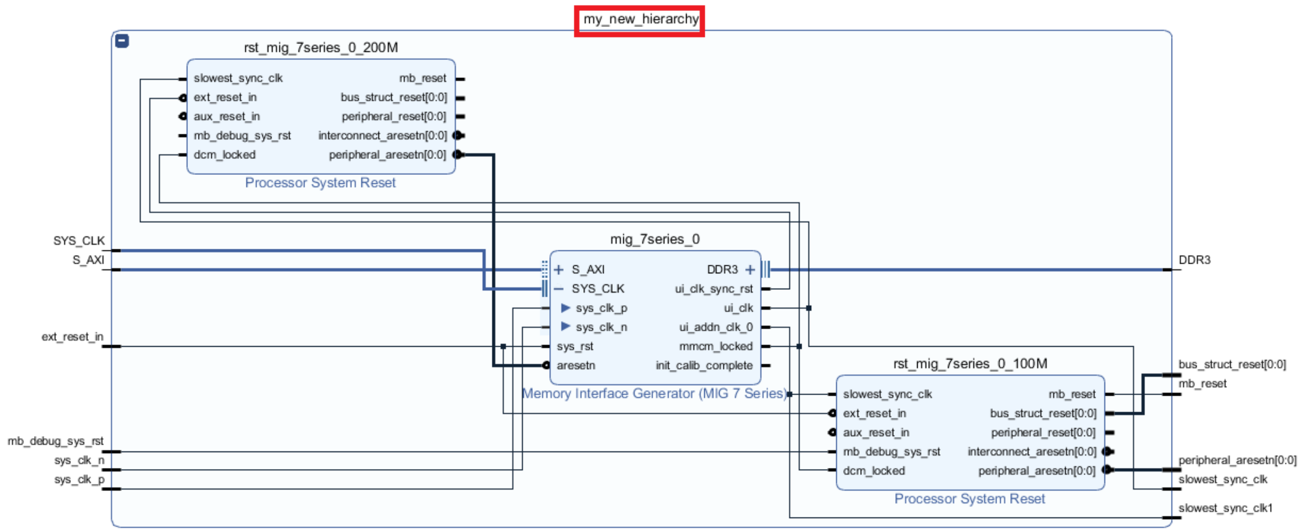
```
##################################################################
# Available Tcl procedures to recreate hierarchical blocks:
#    create_hier_cell_hier_mig parentCell nameHier
##################################################################
```

Now, use the template suggested above in the Tcl Console:

```
create_hier_cell_hier_mig / my_new_hier
```

And the new hierarchical block, called `my_new_hier`, is created in the block design as shown in the following figure.

Figure 154: **Exported Block Design**



The Tcl script created using the `write_bd_tcl` command can then be sourced in a project to re-create the block design by typing source <path_to_tcl_file><filename>.tcl.

If custom IP are present in the block design, the Tcl script created using `write_bd_tcl` contains a pre-check to ensure that the IP repository containing the custom IP have been added to the project prior to creating the block design. If the custom IP repository is not added to the project then the error message similar to the following will be seen when the Tcl file is sourced:

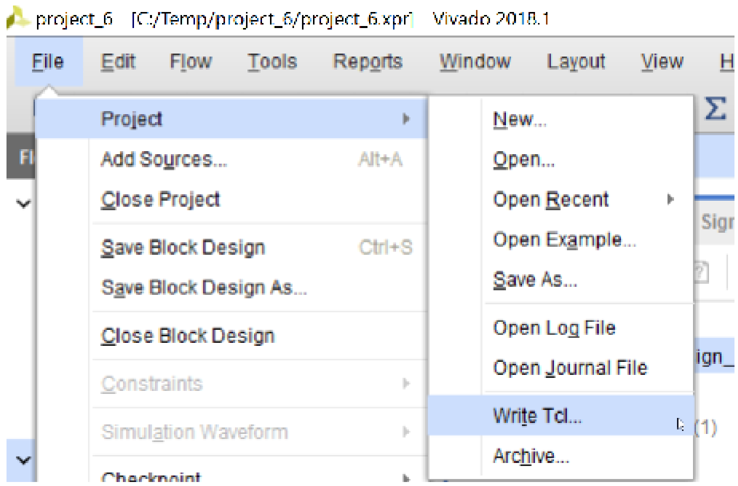
```
ERROR: [BD_TCL-115] The following IPs are not found in the IP Catalog:
  xilinx.com:user:config_mb:1.0

Resolution: Please add the repository containing the IP(s) to the project.
```

As per the error message, the IP repository should be added to the Project before sourcing this Tcl file. IP Repository can be added as described at this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Saving Vivado Project Information in a Tcl File

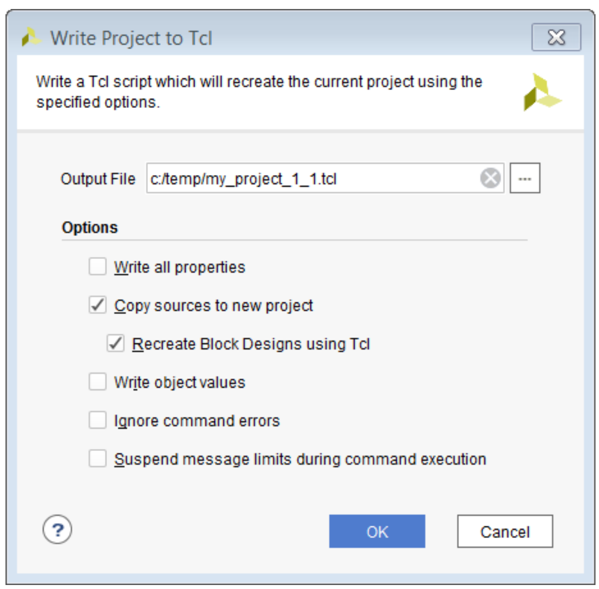
Overall project settings can be saved by selecting File > Project >Write Tcl.

Figure 155: **Writing a Tcl File for the Project**



In the Write Project to Tcl dialog box, shown in the following figure, specify the name and location of the Tcl file and select any other options.

Figure 156: **Write Project to Tcl Dialog Box with Tcl for Block Design**

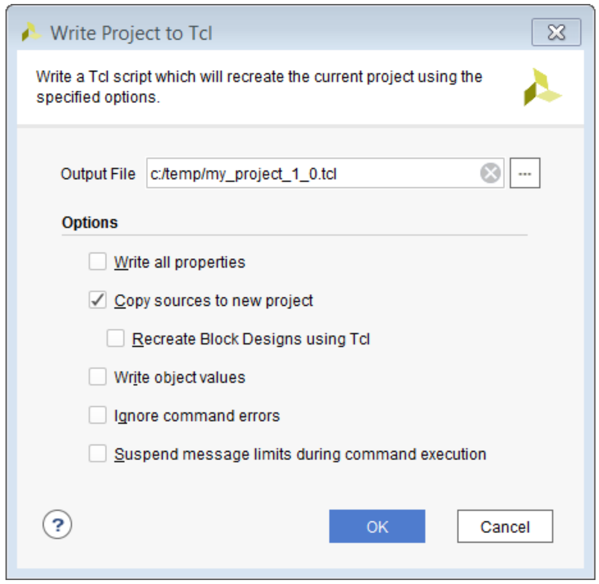


There are several options available in this dialog box. If the Copy sources to new project and the Recreate Block Design using Tcl options are both checked, then the Tcl script to create the project as well as the script to recreate the block design are all included in the script. The same can be done by using the write_project_tcl command in the Tcl Console, as follows:

```
write_project_tcl <path to file>/<filename>.tcl
```

You can also check the option Copy sources to new project without checking the Recreate Block Design using Tcl. In this case, the block design is imported from the original design and added to the project. In other words, the block design is copied from the original design and added to the sources of the project.

Figure 157: **Write Project to Tcl Dialog Box with options to copy block design**

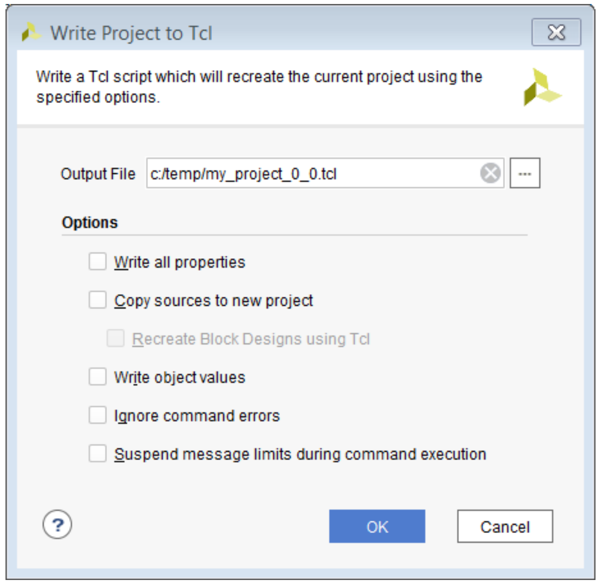


The same can be done by using the `write_project_tcl` command in the Tcl console, as follows:

```
write_project_tcl -use_bd_files {<path to file>/<filename>.tcl}
```

Finally, you can create the Tcl script for the project with both Copy sources to new project and Recreate Block Design using Tcl options unchecked. In this case, the project is created as specified; however, the block design sources are not copied into the project. The block design is rather a "remote" block design, meaning that it points to the block design in the original project.

Figure 158: **Write Project to Tcl Dialog Box with option to use the "remote" block design**



This can also be achieved by typing the following Tcl command in the Tcl console:

```
write_project_tcl -no_copy_sources -use_bd_files {<path to file>/
<filename>.tcl}
```

Chapter 9

Using IP Integrator in Non-Project Mode

Non-Project Mode is for users who want to manage their own design data and manually track the design state. In this mode, Vivado® tools read the various source files and implement the design in-memory throughout the entire design flow. At any stage of the implementation process, you can generate a variety of reports to examine the state of your design.

When running in Non-Project Mode, it is also important to note that the Vivado tool does not enable project-based features such as: source file and design run management, out-of-context (OOC) synthesis, cross-probing back to source files, and design state reporting. Essentially, each time a source file is updated on the disk, you must know about it and reload the design. There are no default reports or intermediate files created within the non-project mode.

You need to have a script to control the creation of reports with Tcl commands. For details of working in non-project mode see this link in *Vivado Design Suite User Guide: Design Flows Overview* (UG892).

Note: If the block design is already generated with one of the out-of-context (OOC) option set, the block design can be added to the non-project flow. If the block design is not generated ahead of adding the block design to the project, you will get an error notifying that the block design must be generated prior to adding it to the non-project flow. If global synthesis option is used, then the block design can be generated within the non-project flow.

Creating a Flow in Non-Project Mode

The recommended approach for running non-project mode is to launch the Vivado Design Suite in Tcl mode, or to create a Tcl script and run the tool in batch mode, using the following command:

```
% vivado -mode batch -source non_project_script.tcl
```

In non-project mode, set your project options as follows:

```
set_part xc7k325tffg900-2
set_property TARGET_LANGUAGE VHDL [current_project]
set_property BOARD_PART xilinx.com:kc705:part0:0.9 [current_project]
set_property DEFAULT_LIB work [current_project]
```

In non-project mode, there is no project file saved to disk. Instead, an in-memory Vivado project is created. The device/part/target-language of a block design is not stored as a part of the block design sources. The `set_part` command creates an in-memory project for a non-project based design, or assigns the part to the existing in-memory project.

After the in-memory project has been created, the source file (.bd) for the block design can be added to the design. This can be done in two different ways:

- First, assuming that there is an existing block design with the output products generated and intact, you can add the block design using the read_bd Tcl command as follows:
-

```
read_bd <path to the bd file>
```

Note: If the block design is not generated then you will need to generate the output products for the block design by adding the following commands:

```
read_bd <path to the bd file>
set_property synth_checkpoint_mode None [get_files <path to the bd file>]
generate_target all [get_files <path to the bd file>]
```

CAUTION! *The settings (board, part, and user repository) of the new design must match the settings of the original block design, or the IP in the block design will be locked.*

After the block design is added successfully, you need to add your top-level RTL files and any top-level XDC constraints. You will also need to instantiate the block design into your top-level RTL.

```
read_verilog <top-level>.v
read_xdc <top-level>.xdc
```

- Second, you can use the block design as the top-level of the design by creating an HDL wrapper file for the block design using the following commands:
-

```
make_wrapper -files [get_files <path to bd>/<bd instance name>.bd] -top
read_vhdl <path to bd>/<bd instance name>_wrapper.vhd
update_compile_order -fileset sources_1
```

This creates a top-level HDL file and adds it to the source list. The top-level HDL wrapper around the block design is needed because a BD source cannot be synthesized directly.

For a MicroBlaze-based processor design, you need to add and associate an ELF with the MicroBlaze instance in the block design. This populates the block RAM initialization strings with the data from the ELF file. You can do this with the following commands:

```
add_files <file_name>.elf
set_property SCOPED_TO_CELLS {microblaze_0} [get_files <file_name>.elf]
set_property SCOPED_TO_REF {<bd_instance_name>} [get_files
<file_name>.elf]
```

TIP: *With the ELF file added to the project, and associated with the processor, the Vivado tools automatically merges the Block RAM memory information (MMI file) and the ELF file contents with the device bitstream (BIT) when generating the bitstream to program the device.*

You can also merge the MMI, ELF, and BIT files after the bitstream has been generated by using the updatemem utility. See this link in the *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898) for more information.

If the design has multiple levels of hierarchy, you need to ensure that the correct hierarchy is provided. After this, go through the usual synthesis, place, and route steps to implement the design.

```
synth_design -top <top module name>
opt_design
place_design
route_design
write_bitstream top
```

To export the implemented hardware system to SDK, use the following command:

```
write_sysdef -hwdef ./<hwdef_file_name>.hwdef \
-bitfile ./<bit_file_name>.bit \
-meminfo ./<mmi_file_name>.mmi \
-file ./<sysdef_file_name>.sysdef
```

You can click the blue, underlined command links to see the write_sysdef or write_hwdef commands in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) for more information on the Tcl commands.

Non-Project Script

The following is a sample script for creating a block design in non-project mode.

```
set_part xc7k325tffg900-2
set_property target_language VHDL [current_project]
set_property board_part xilinx.com:kc705:part0:0.9 [current_project]
set_property default_lib work [current_project]
read_bd ./bd/mb_ex_1/mb_ex_1.bd
open_bd_design ./bd/mb_ex_1/mb_ex_1.bd
```

```
read_vhdl  ./bd/mb_ex_1/hdl/mb_ex_1_wrapper.vhd
write_hwdef -file mb_ex_1_wrapper.hwdef
set_property source_mgmt_mode All [current_project]
update_compile_order -fileset sources_1
```

Updating Designs for a New Release

As you upgrade your Vivado® Design Suite to the latest release, the recommended flow is to upgrade the block designs created in the Vivado® IP Integrator as well.

- The IP version numbers can change from one release to another.
- When IP Integrator detects that the IP contained within a block design are older versions of the IP, it *locks* those IP in the block design.

If the intention is to keep older version of the block design (and the IP contained within it), then you do not need to do any operations such as modifying the block design on the canvas, validating it and/or resetting output products, and re-generating output products. In this case, the expectation is that you have all the design data from the previous release intact. You can use the block design from the previous release "as is" by synthesizing and implementing the design.

It is also possible to selectively upgrade only some of the IP inside the block design. Please see Selectively Upgrading IP in Block Designs.

The recommended practice is to upgrade the block design with the latest IP versions, make any necessary design changes, validate design and generate target.

You can update projects in two ways:

- Upgrading a Block Design in Project Mode
- Upgrading a Block Design in Non-Project Mode

This chapter describes both methods.

Upgrading a Block Design in Project Mode

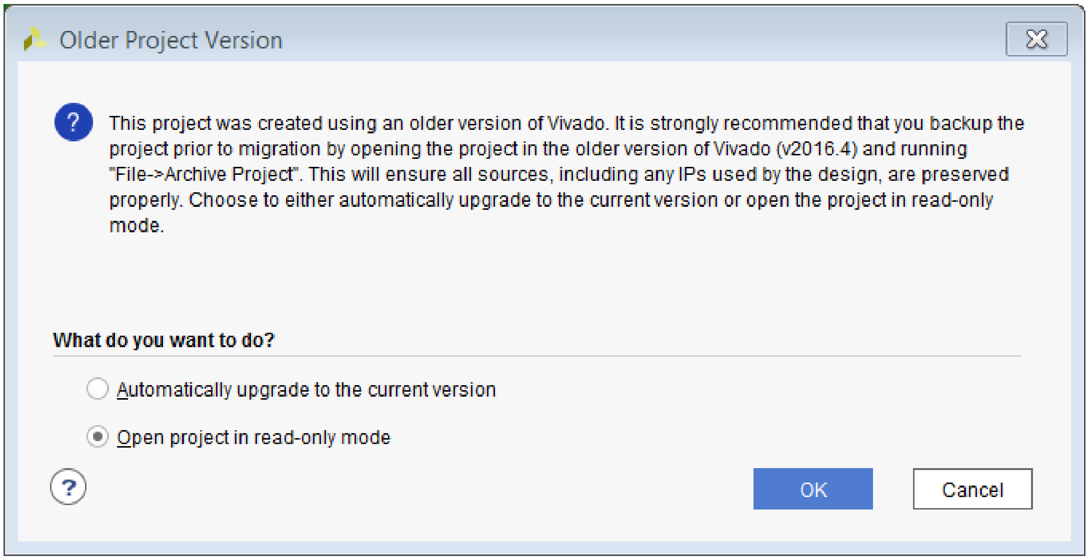
To upgrade a block design in project mode:

1. Launch the latest version of the Vivado Design Suite.
2. From the Vivado IDE, click **File→Project→Open**, and navigate to the design that was created from a previous version of Vivado tools.

The Older Project Version pop-up opens. Automatically upgrade to the current version is selected by default.

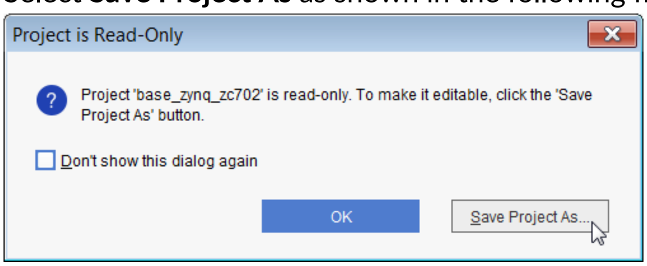
Although you can upgrade the design from a previous version by selecting the Automatically upgrade to the current version, it is highly recommended that you save your project with a different name before upgrading.

3. Select **Open project in read-only mode**, as shown in the following figure, and click **OK**.

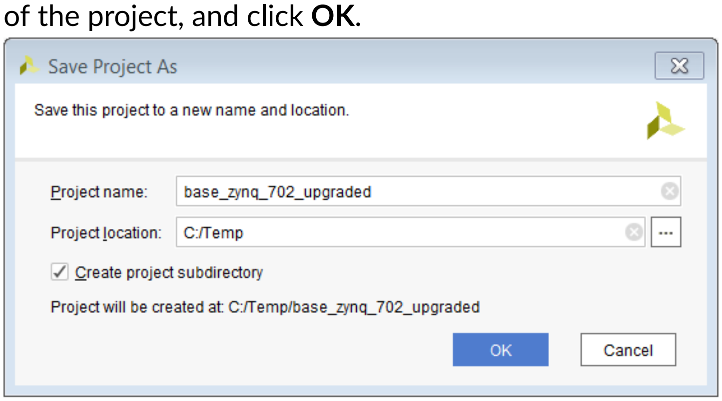


The Project is Read-Only dialog box opens.

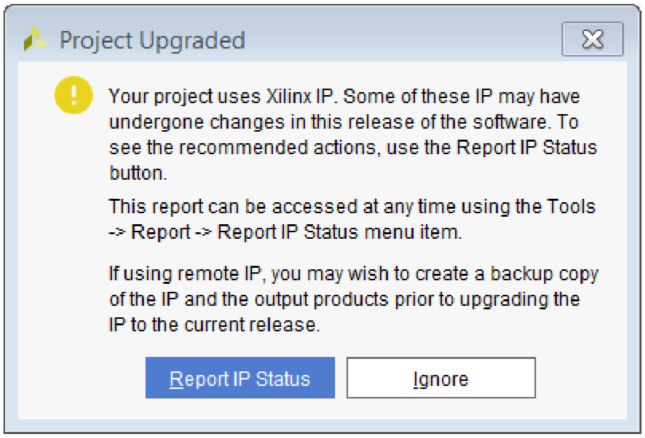
4. Select **Save Project As** as shown in the following figure.



5. When the Save Project As dialog box opens, as shown in the following figure, type the name of the project, and click **OK**.



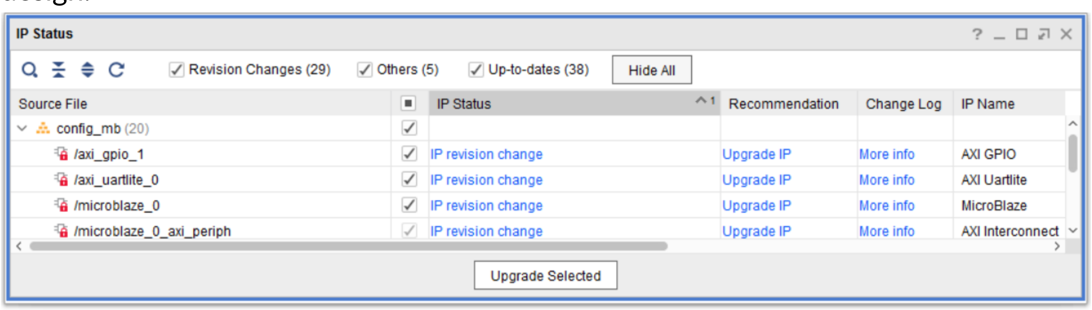
The Project Upgraded dialog box opens, as shown in the following figure, informing you that the IP used in the design may have changed and therefore need to be updated.



6. Click **Report IP Status**.

   Alternatively, from the menu, select **Reports → Report IP Status**.

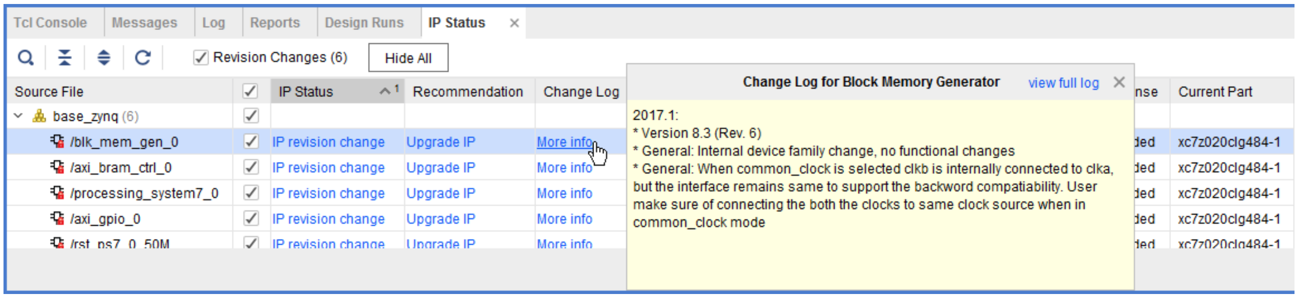
7. In the IP Status window, look at the different columns and familiarize yourself with the IP Status report. Expand the block design and look at the changes of IP cores in the block design.



The top of the IP Status window shows the summary of the design. It reports how many changes are needed to upgrade the design to the current version. The changes reported are Major Changes, Minor Changes, Revision Changes, and Other Changes. These changes are reported in the IP Status column as well.

- **Major Changes:** The IP has gone through a major version change; for example, from Version 2.0 to 3.0. This type of change is not automatically selected for upgrade. To select this for upgrade, uncheck the Upgrade column for the block design and then re-check it.
- **Minor Changes:** The IP has undergone a minor version change; for example, from version 3.0 to 3.1.
- **Revision Changes:** A revision change has been made to the IP; for example, the current version of the IP is 5.0, and the upgraded version is 5.0 (Rev. 1).

8. To see a description of the change, click **More info** in the Change Log column, shown in the following figure.



The Recommendation column also suggests that you need to understand what the changes are before selecting the IP for upgrade.

9. When you understand the changes and the potential impact on your design, click **Upgrade Selected**.

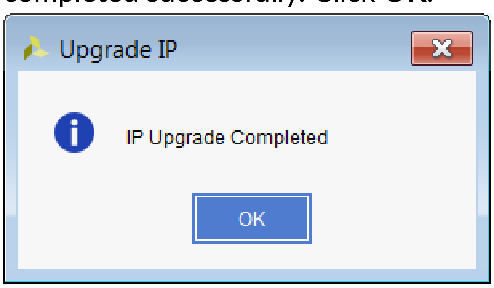
   The Upgrade IP dialog box opens to confirm that you want to proceed with upgrade.

10. Click **OK**.

    When the upgrade process is complete, a Critical Messages dialog box may open, informing you about any critical issues to which you need to pay attention. Review any critical warnings and other messages that may be flagged as a part of the upgrade.

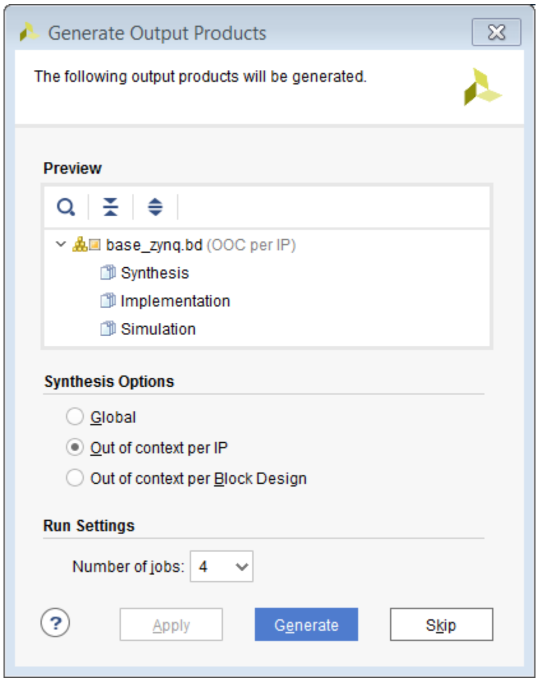
11. Click **OK**.

12. If there are no Critical Warnings, the Upgrade IP dialog box informs you that the IP Upgrade completed successfully. Click **OK**.



Regenerating Output Products

1. The Generate Output Product dialog box opens. You can skip generation at this time by clicking Skip or click Generate to generate the block design.



2. You can now synthesize, implement, and generate the bitstream for the design.

Upgrading a Block Design in Non-Project Mode

You can open an existing project from a previous release using the non-project mode flow and upgrade the block design to the current release of Vivado. However, if out-of-context (OOC) mode is used for the block design, then the block design must be upgraded and generated in a project mode ahead of adding the block design to the non-project flow. Use the following script as a guideline to upgrade the IP in the block diagram (applicable only when the block design is synthesized with the Global synthesis option):

```
# Open an existing project from a previous Vivado release
open_project <path_to_project>/project_name.xpr
update_compile_order -fileset sim_1
# Open the block diagram
read_bd {<path_to_bd>/bd_name.bd}
# Make the block diagram current
current_bd_design bd_name.bd
# Upgrade IP
upgrade_bd_cells [get_bd_cells -hierarchical * ]
# Reset output products
reset_target {synthesis simulation implementation} [get_files
<path_to_project>/project_name.srcs/sources_1/bd/bd_name/bd_name.bd]
# Generate output products
generate_target {synthesis simulation implementation} [get_files
<path_to_project>/project_name/project_name.srcs/sources_1/bd/bd_name/
```

```
bd_name.bd]
# Create HDL Wrapper (if needed)
make_wrapper -files [get_files
<path_to_project>/project_name/project_name.srcs/sources_1/bd/bd_name/
bd_name.bd]
-top
# Overwrite any existing HDL wrapper from before
import_files -force -norecurse
<path_to_project>/project_name/project_name.srcs/sources_1/bd/bd_name/hdl/
bd_name_w
rapper.v
update_compile_order -fileset sources_1
# Continue through implementation
```

Selectively Upgrading IP in Block Designs

Often times users want their IP that have been validated in the previous release to be not upgraded. It is possible to selectively upgrade some IP within a block design. However, there are some limitations to this flow that a user must understand. The process to selectively upgrade IP, the requirements, the consequences of doing so and the limitations to this flow are described in this section.

Requirements of Selectively Upgrading IP in Block Designs

The main requirement of using this feature is that the block design must be fully generated in the previous release of Vivado. If the block design is not fully generated, then this feature cannot be used.

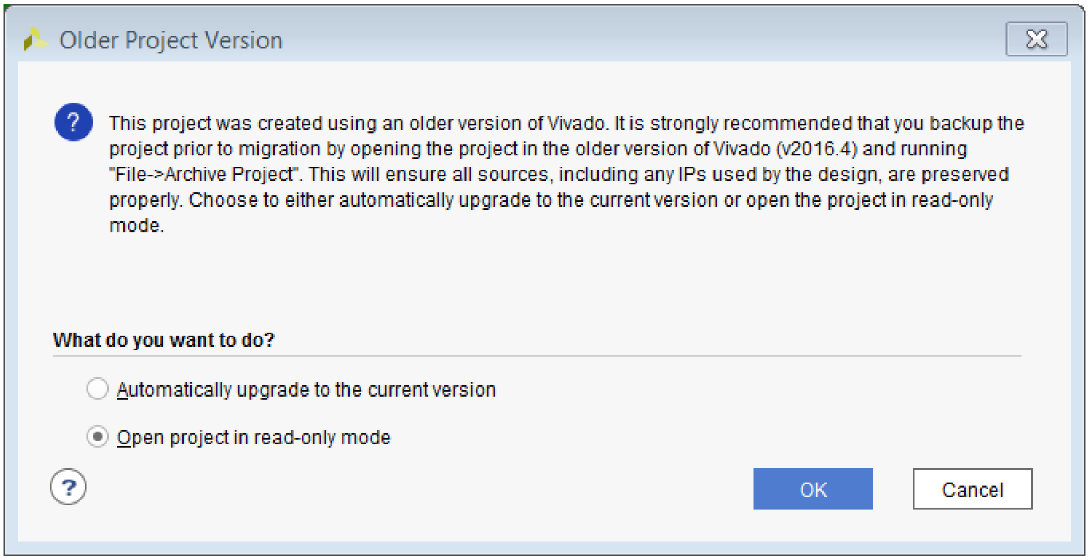
Selectively Upgrading IP Flow in Project Mode

To upgrade a block design using the Selective IP upgrade feature in project mode:

1. Launch the latest version of the Vivado Design Suite.
2. From the Vivado IDE, click **File → Project → Open**, and navigate to the design that was created from a previous version of Vivado tools.

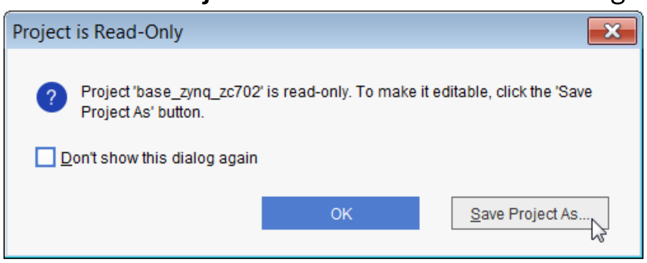
   The Older Project Version pop-up opens. Automatically upgrade to the current version is selected by default.

   Although you can upgrade the design from a previous version by selecting the Automatically upgrade to the current version, it is highly recommended that you save your project with a different name before upgrading. To do so:
3. Select **Open project in read-only mode**, as shown in the following figure, and click **OK**.

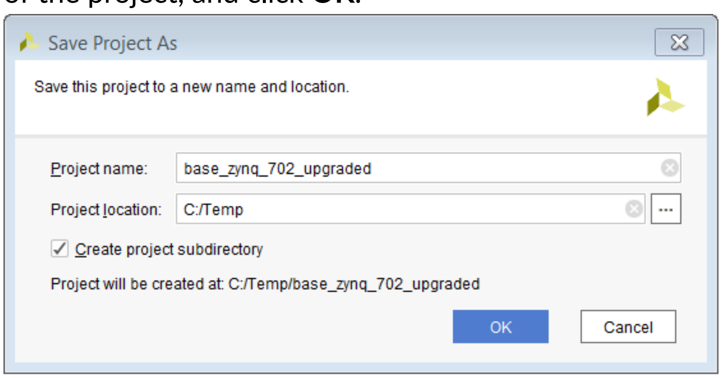


The Project is Read-Only dialog box opens.

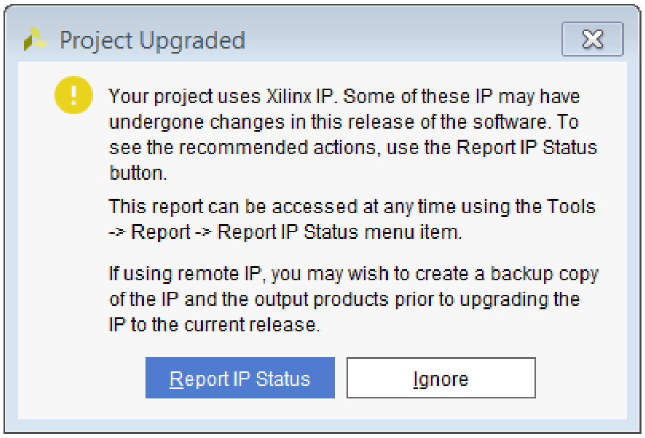
4. Select **Save Project As** as shown in the following figure.



5. When the Save Project As dialog box opens, as shown in the following figure, type the name of the project, and click **OK**.



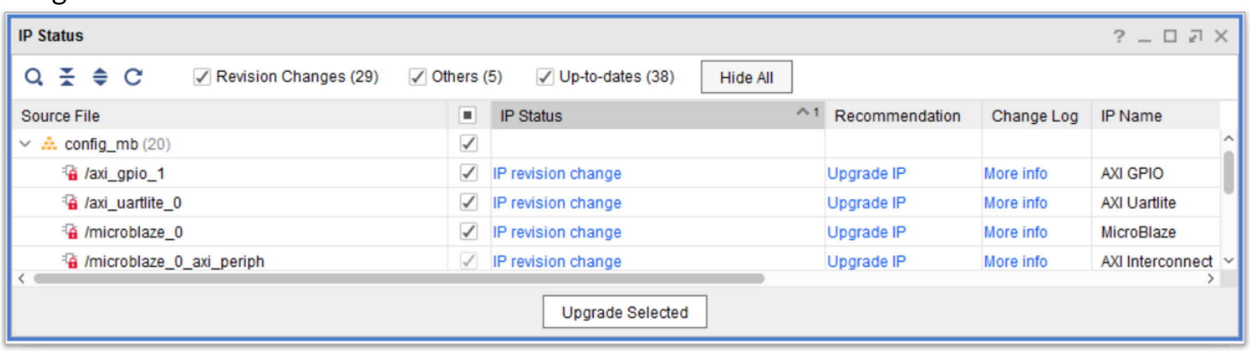
The Project Upgraded dialog box opens, as shown in the following figure, informing you that the IP used in the design could have changed and therefore need to be updated.



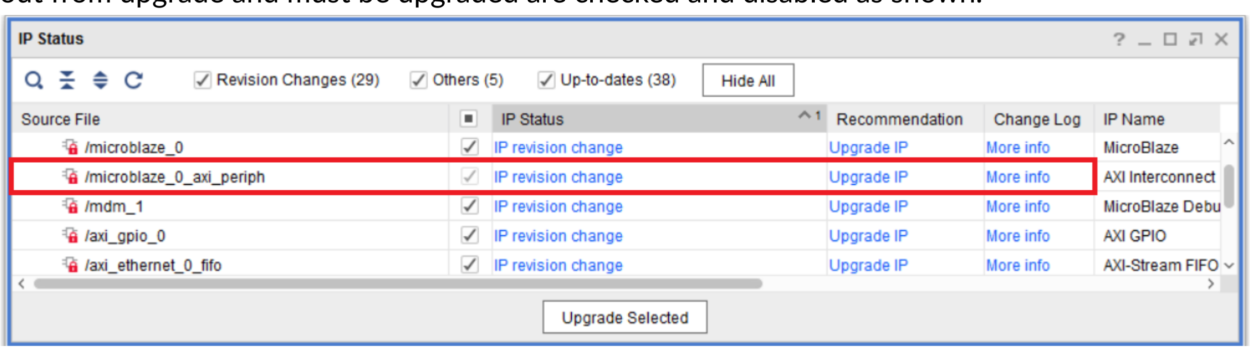
6. Click **Report IP Status**.

   Alternatively, from the menu select **Reports → Report IP Status**.

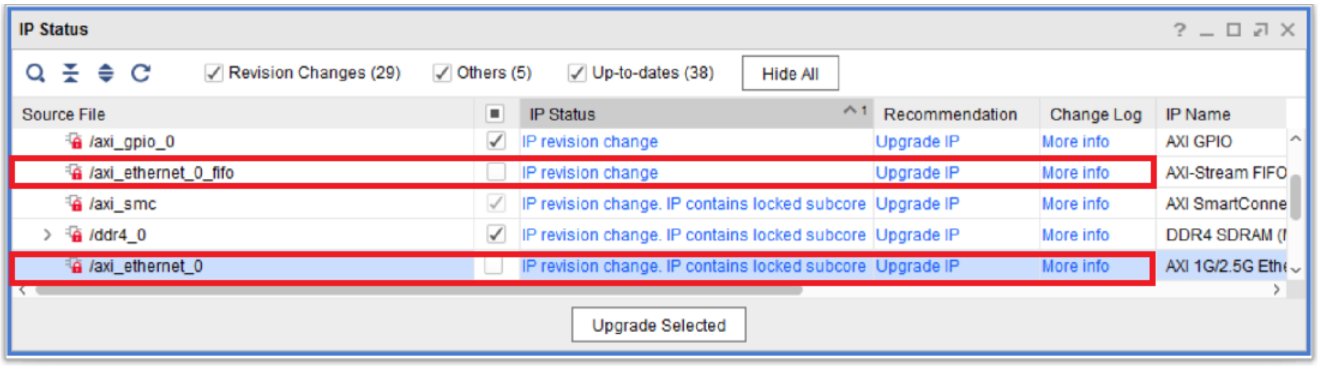
7. In the IP Status window, look at the different columns and familiarize yourself with the IP Status report. Expand the block design and look at the changes of IP cores in the block design.



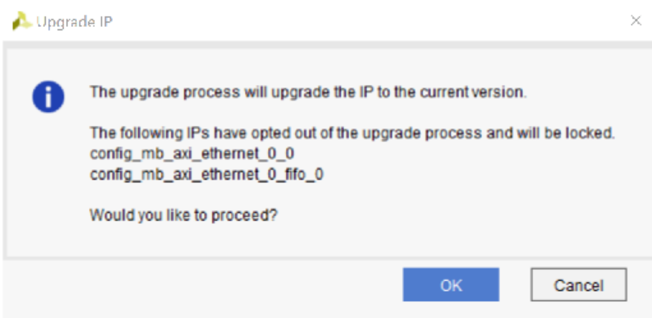
8. All the IP in the block design are selected by default for upgrade. The IP that cannot be opted out from upgrade and must be upgraded are checked and disabled as shown.



9. Uncheck any IP that need not be upgraded.



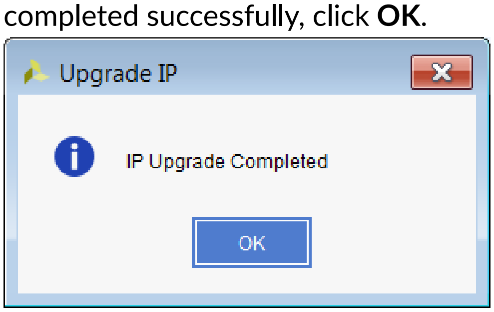
10. Click **Upgrade Selected**.
11. The Upgrade IP dialog box opens to confirm that the checked IP in the IP Status window will be upgraded and provides a list of the unchecked IP that will not be upgraded.



12. Click **OK**.

    When the upgrade process is complete, a Critical Messages dialog box may open, informing you about any critical issues to which you need to pay attention. Review any critical warnings and other messages that may be flagged as a part of the upgrade.

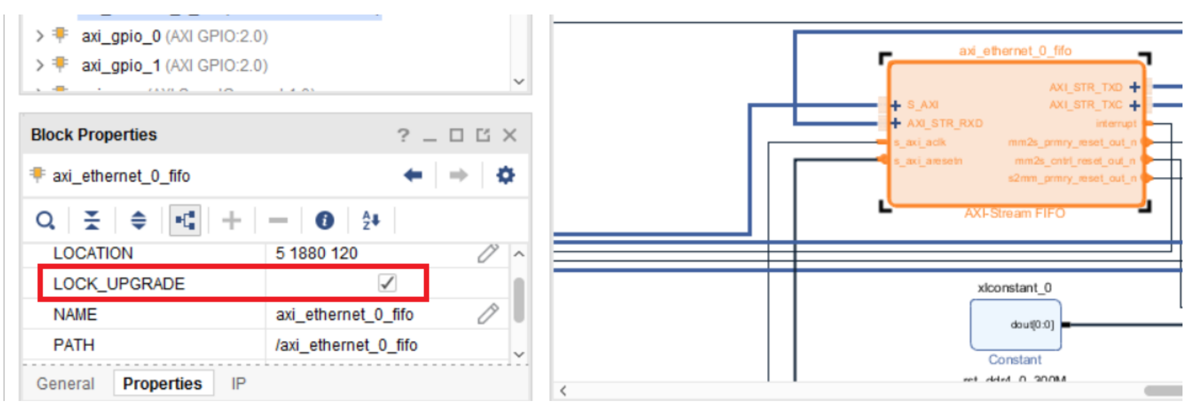
13. Click **OK**.
14. If there are no Critical Warnings, the Upgrade IP dialog box informs you that the IP Upgrade completed successfully, click **OK**.



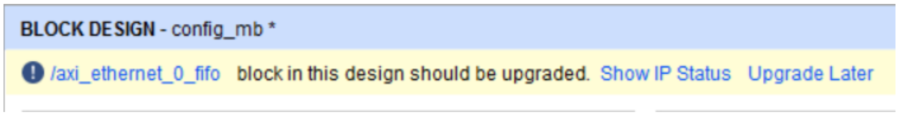
When the IP are upgraded or locked, a couple of things are worth noticing. As highlighted in the messages from the Tcl console, first of all the IP that have been opted out of upgrade, will have a special property called `LOCK_UPGRADE` set on them. The rest of the IP in the design are upgraded as in the normal flow.

```
set_property LOCK_UPGRADE 1 [get_bd_cells /axi_ethernet_0]
WARNING: [BD 41-2028] Locking axi_ethernet to version 7.1. If the latest
driver for
the IP is not backwards compatible, software flow will assign a generic
driver or no
driver for this IP
INFO: [Vivado 12-5777] IP Instance 'config_mb_axi_ethernet_0_0' cannot
be used in a
module reference: The 'xilinx.com:ip:axi_ethernet:7.1' core does not
support module
reference.
set_property LOCK_UPGRADE 1 [get_bd_cells /axi_ethernet_0_fifo]
WARNING: [BD 41-2028] Locking axi_fifo_mm_s to version 4.1. If the
latest driver for
the IP is not backwards compatible, software flow will assign a generic
driver or no
driver for this IP
INFO: [Vivado 12-5777] IP Instance 'config_mb_axi_ethernet_0_0' cannot
be used in a
module reference: The 'xilinx.com:ip:axi_ethernet:7.1' core does not
support module
reference.
upgrade_ip [get_ips  {config_mb_axi_gpio_0_0 config_mb_axi_uartlite_0_0
config_mb_microblaze_0_axi_periph_0 config_mb_ddr4_0_0
config_mb_axi_gpio_1_0
config_mb_xlconstant_0_0 config_mb_microblaze_0_0 config_mb_axi_smc_0
config_mb_mdm_1_0}] -log ip_upgrade.log
Upgrading
'C:/bash/2018.1/selective_upgrade/project_4/project_4.srcs/sources_1/bd/
config_mb/c
onfig_mb.bd'
WARNING: [Vivado 12-3647] The given sub-design is not contained in the
block fileset
'config_mb_axi_gpio_0_0'. Sub-design: Regenerating Output Products
```

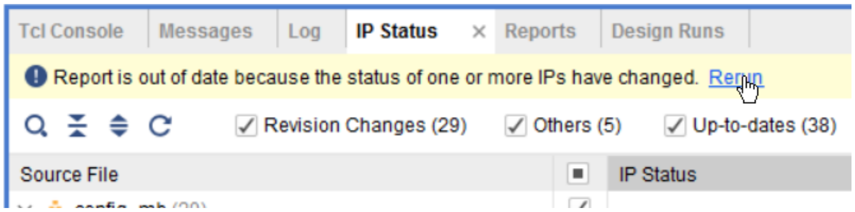
The property `LOCK_UPGRADE` can also be seen by selecting the cell in the block design canvas and then looking at the Properties window.



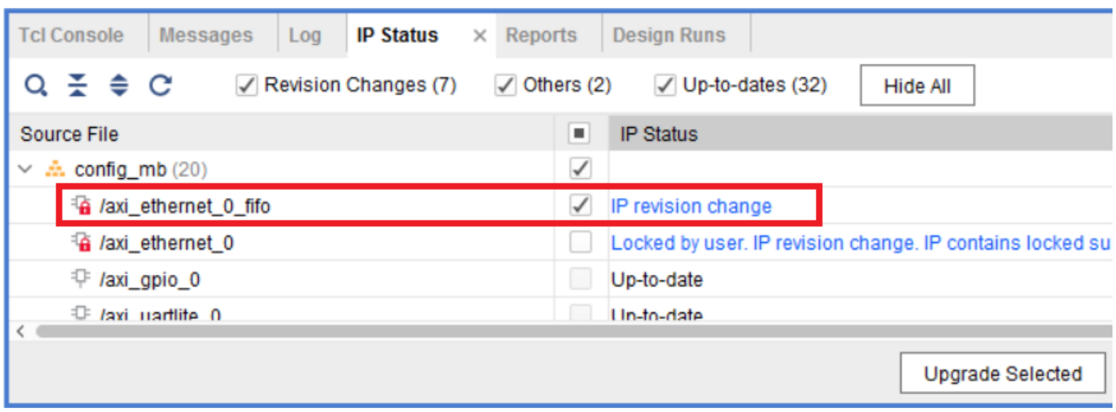
This property can be toggled through the Properties window as well. When a user toggles this property through the Properties window, the banner in the block design canvas, prompts the user to upgrade the design.



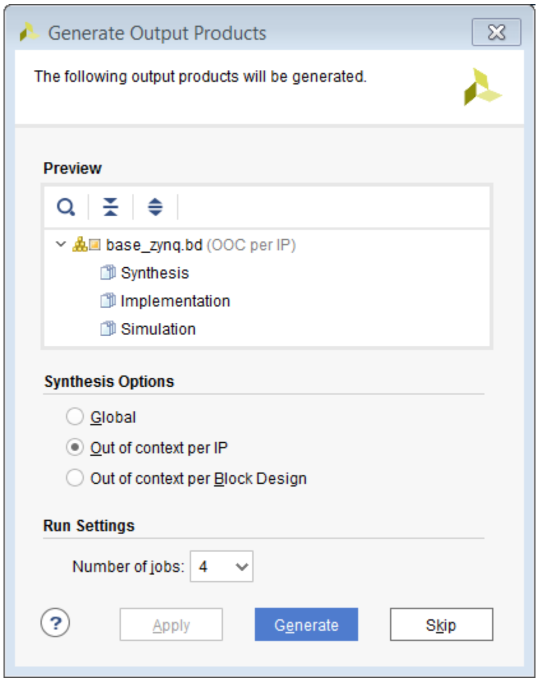
Clicking on Show IP Status link will bring the IP Status window view. Click on Rerun to refresh the status of IP in the IP Status window.



The refreshed IP Status window shows that because you chose to toggle the `LOCK_UPGRADE` property in the Properties window for the axi_ethernet_0_fifo IP, that IP needs to be upgraded. The IP is checked for upgrade as shown below. Update the IP by clicking the Upgrade Selected button.



15. The Generate Output Product dialog box opens. You can skip generation at this time by clicking **Skip** or click **Generate** to generate the block design.



16. You can now synthesize, implement, and generate the bitstream for the design.

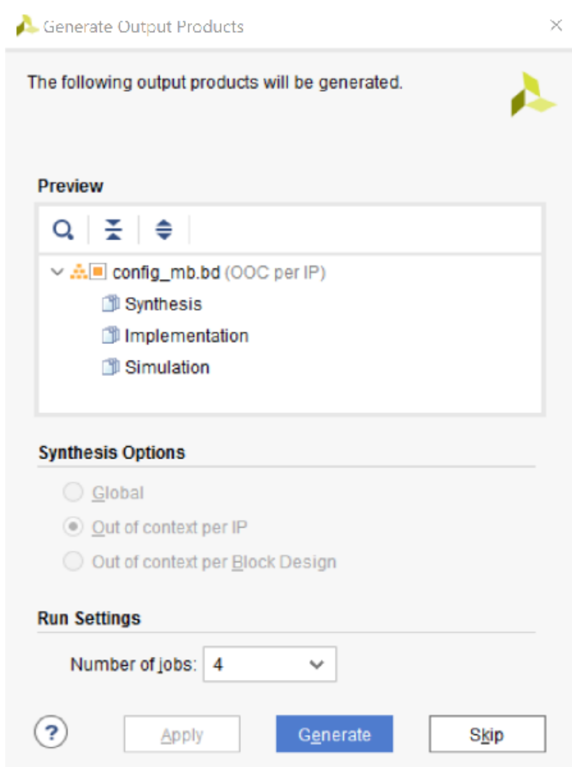
Limitations of Selectively Upgrading IP in Block Designs

The following are the limitations of this feature:

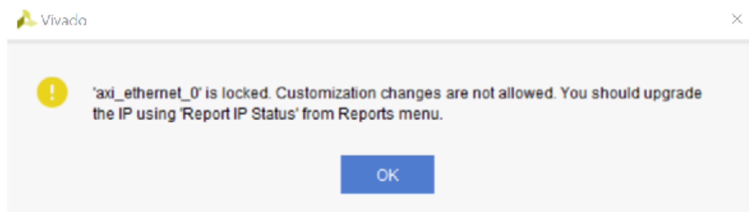
- The following IP are not supported in this feature. If these IP are used in a block design, they must be upgraded when migrating from an older release of Vivado.
- AXI SmartConnect
- AXI Interconnect
- AXI4-Stream Interconnect
- System ILA
- Zynq UltraScale+ MPSoC
- Zynq
- ILA (Integrated Logic Analyzer)
- Debug Bridge
- VIO (Virtual Input/Output)
- JTAG To AXI Master
- Video Test Pattern Generator
- JESD204 PHY

- JESD204
- Gmii to Rgmii
- HDMI 1.4/2.0 Receiver Subsystem
- HDMI 1.4/2.0 Transmitter Subsystem
- Video PHY Controller
- Video Frame Buffer Read
- Video Frame Buffer Write
- MIPI CSI-2 RX Subsystem
- Block Memory Generator
- RTL Modules
- The synthesis mode cannot be changed when using this feature. As an example, if the synthesis mode selected in the previous release of Vivado was Out-of-context per IP, then this mode cannot be changed to Global or Out-of-context per Block Design. In the GUI, the synthesis options cannot be changed as shown below.

Figure 159: **Generate Output Products Dialog Box with Synthesis Options Disabled**



- The IP that have been chosen to be not upgraded or locked, cannot be parameterized. They cannot participate in parameter propagation. In other words, because the parameters of the IP are locked, they cannot be changed by other IP in the design. However, if the IP propagates parameters to other IP within the design, the parameters will be propagated.



- The Tcl script for a block design containing locked IP cannot be generated using `write_bd_tcl`. If the user tries to do so, the following error message will be flagged.
-
```
write_bd_tcl -force ./selective_upgrade/conf_mb_des_fg/config_mb.tcl
ERROR: [BD 5-599] write_bd_tcl is not supported for block design with IP
that have
not been upgraded to their latest version. Please upgrade all the IP to
their latest
version.
ERROR: [Common 17-39] 'write_bd_tcl' failed due to earlier errors.
```
- User locked IP cannot be copied and pasted in the block design canvas.
- Copying a block design that has locked IP using the command File > Save Block Design As cannot be done. If the user chooses to perform this action, the following error message will be flagged.
-
```
ERROR: [BD 41-1179] The following IP in this design are locked. This
command cannot
be run until these IP are unlocked. Please run report_ip_status for more
details and
a recommendation on how to fix this issue.
/axi_ethernet_0
/axi_ethernet_0_fifo

ERROR: [Common 17-39] 'save_bd_design_as' failed due to earlier errors.
```

Using the Platform Board Flow in IP Integrator

The Vivado® Design Suite is board aware. The tools know the various interfaces present on the target boards and can customize and configure an IP to be connected to a particular board component. Several standard 7 series boards are available in the Vivado Design Suite, as well as an UltraScale® architecture board.

You also have the ability to define custom board files to add to the tool. See this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) for more information on the Board Interface file.

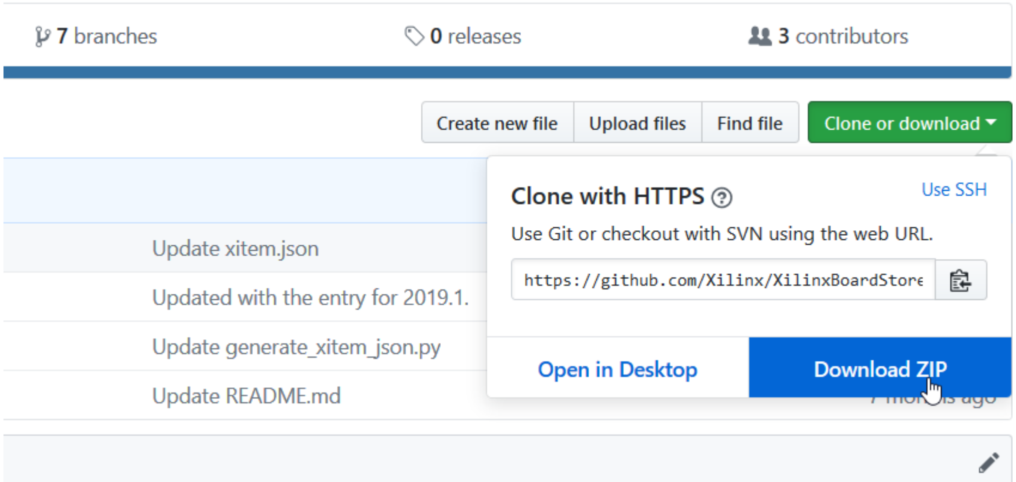
The IP integrator shows all the interfaces to the board in a separate Board window. When you use this window to select components and the Designer Assistance offered by IP integrator, you can easily connect your block design to the board components of your choosing. This flow generates all the I/O constraints automatically.

User-defined or third-party Board Interface files, and associated files, can be downloaded from GitHub at https://github.com/Xilinx/XilinxBoardStore/wiki/Xilinx-Board-Store-Home.

From the Xilinx Board Store page:

1. Click **Code**, as shown in the following figure.

2. Click the **Clone** or download pull-down button to open the Clone with HTTPS dialog box.
3. Click **Download ZIP**.



4. Save the boards at a location of your choice.

These boards can then be added to a board repository for use by the Vivado Design Suite by setting the following parameter when launching the Vivado tool:

```
set_param board.repoPaths [list “<path1>” “<path2>” “...”]
```

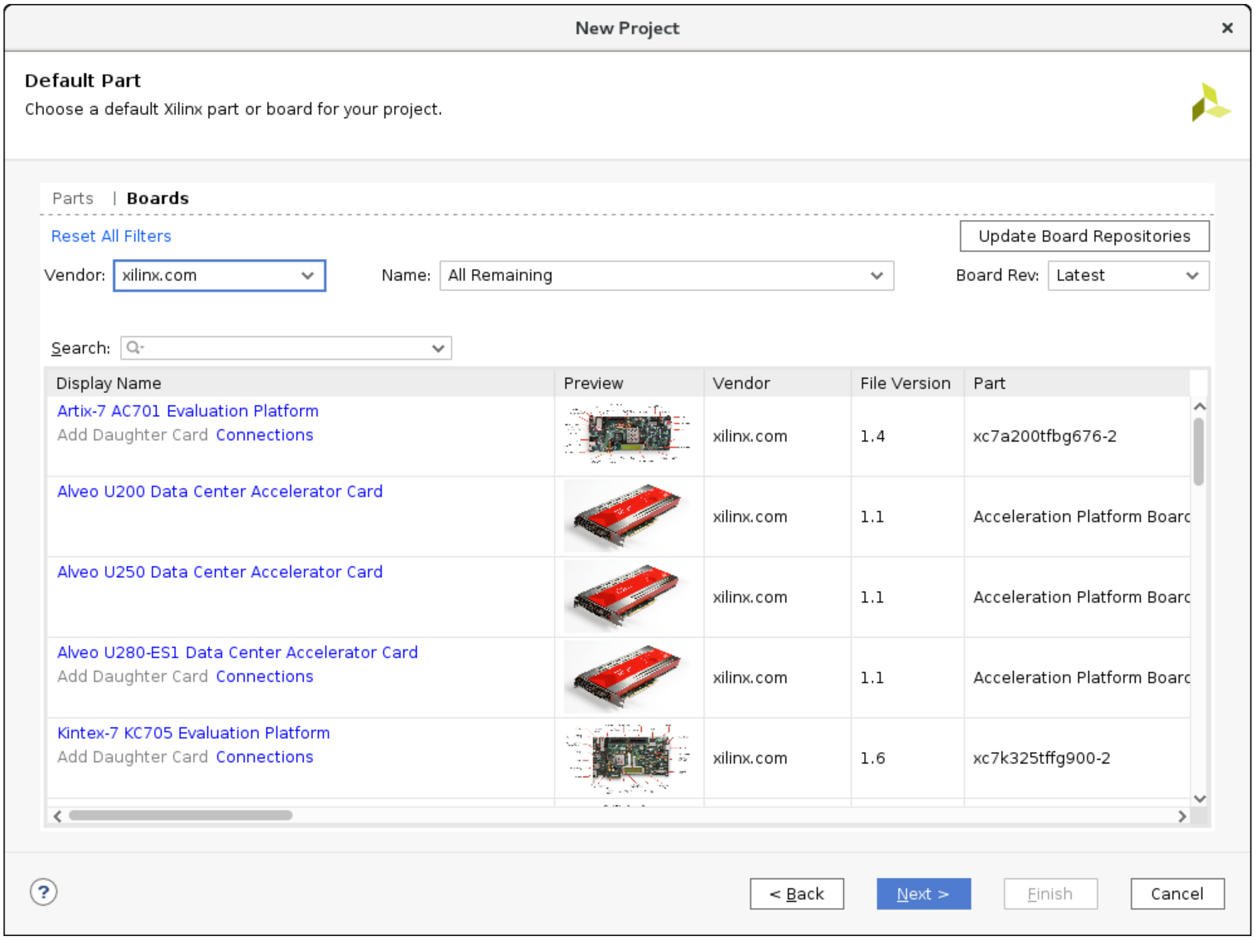
Where `<path>` is the path to a directory containing a single Board Interface file, and files referenced by the board.xml file, such as part0_pins.xml and preset.xml. The `<path>` can also specify a directory with multiple subdirectories, each containing a separate Board Interface file. For example:

```
set_param board.repoPaths [list "C:/Data/usrBrds" "C:/Data/othrBrds"]
```

Selecting a Target Board

When a new project is created in the Vivado environment, you have the option to select a target board from the Default Part page of the New Project wizard. From this dialog box user-defined or third-party Board Interface files, and associated files, can be downloaded from GitHub as well. By default, only boards support natively in Vivado are shown in the dialog box.

Figure 160: **Select a Target Board**



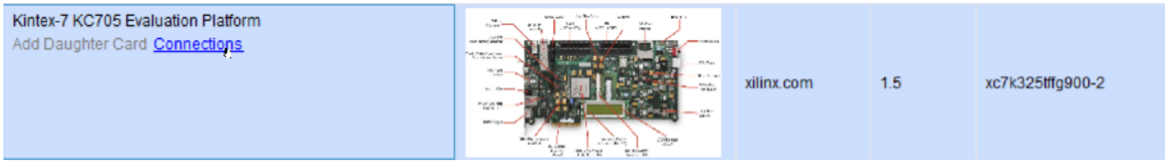
You can filter the list of available boards based on Vendor, Display Name, and Board Revision.

- Vendor: Specifies the board manufacturer.
- Name: Lists the name for the board.
- Board Rev: Allows filtering based on the revision of the board. Setting the Board Rev to All shows revisions of all the boards that are supported in Vivado. Setting Board Rev to Latest shows only the latest revision of a target board.
- Various information such as resources available and operating conditions are also listed in the table.

When you select a board, the project is configured using the pre-defined interface for that board.

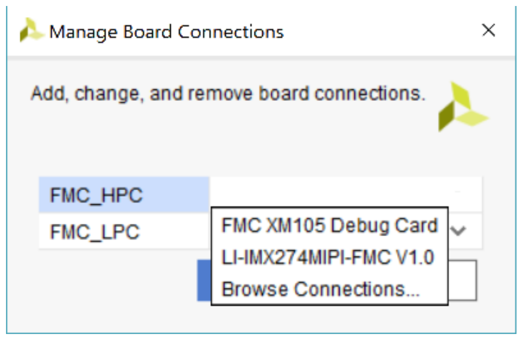
Some boards also support connections to the FMC connectors present on them. In such cases, there is a link to add a Daughter Card to the board as shown below. (KC705 board is shown.)

Figure 161: **Check to see if a daughter card exists for the target board**



Click the Connections link to bring the dialog box to see a list of the available FMC cards.

Figure 162: **Select a Daughter Card**

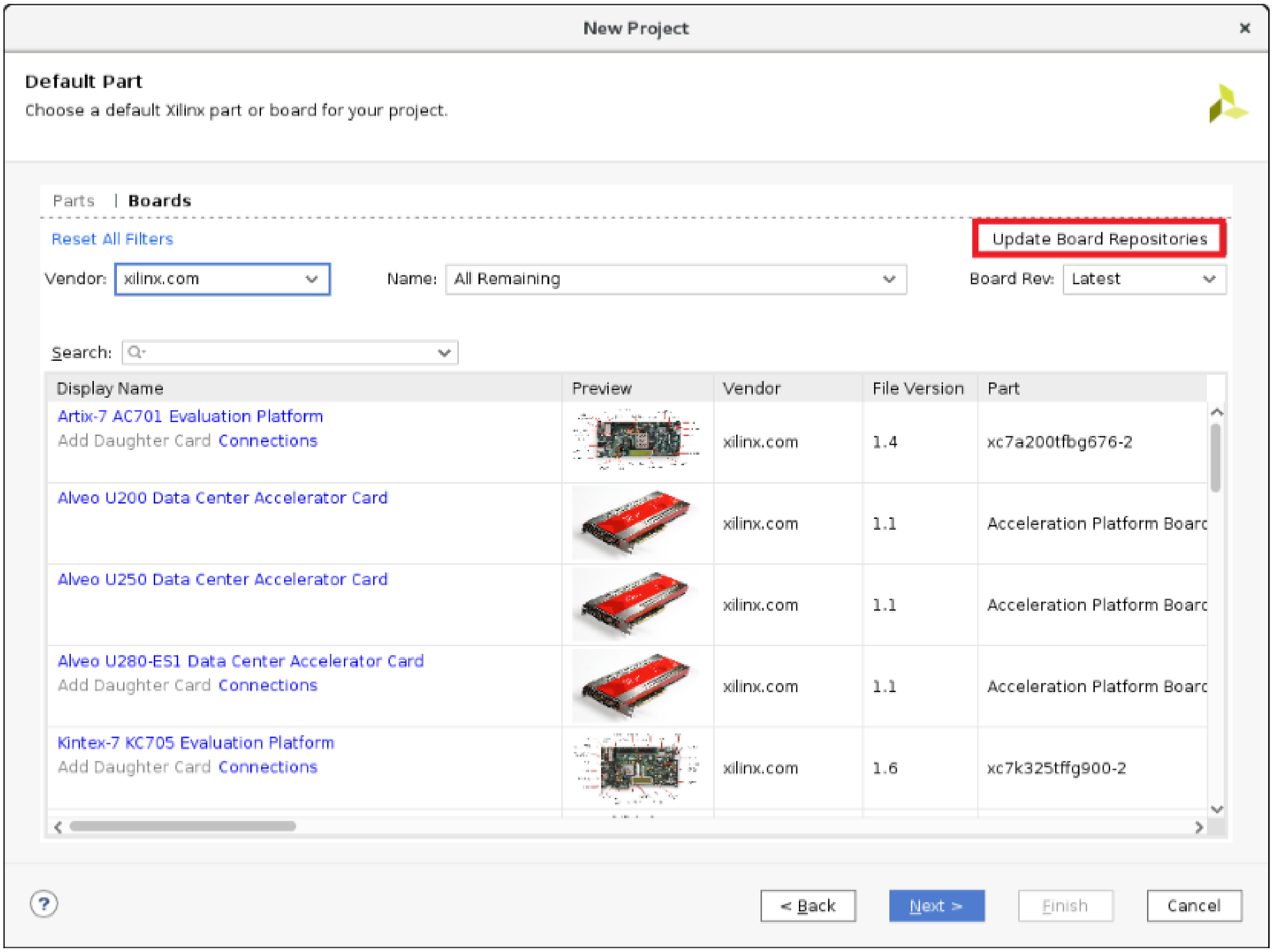


As shown, when KC705 is selected, the FMC_HPC and FMC_LPC connectors show the available FMC cards. Selecting a card enables all the connections on that card available for use in the design.

Downloading Third-Party board files from Git Hub using the GUI

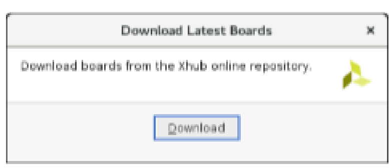
Third party board files or even Xilinx provided board files that are not natively supported in Vivado can be downloaded from the Git Hub using the GUI. The Default Boards Page of the New Project Wizard has a mechanism to download the board files available on the Git Hub. To download boards click **Update Board Repositories** on the top right corner of the Default Part page.

Figure 163: **Download third-party board files from Git Hub**



The **Download Latest Boards** dialog box opens. Click **Download**.

Figure 164: **Download Latest Boards**



The first time you download the boards from the Git Hub you will download the entire repository available. Subsequent downloads will only download the boards that are new and have not been previously downloaded. As the boards are being downloaded you will see messages such as the following on the TCL console.

```
INFO: [Common 17-1570] Installing object
em.avnet.com:xilinx_board_store:Ultra96:1.2 from remote host https://
github.com/Xilinx/XilinxBoardStore.git
INFO: [Common 17-1573] Object em.avnet.com:xilinx_board_store:Ultra96:1.2
has been installed successfully.
INFO: [Common 17-1570] Installing object
em.avnet.com:xilinx_board_store:picozed_7010_fmc2:1.2 from remote host
https://github.com/Xilinx/XilinxBoardStore.git
INFO: [Common 17-1573] Object
em.avnet.com:xilinx_board_store:picozed_7010_fmc2:1.2 has been installed
successfully.
INFO: [Common 17-1570] Installing object
digilentinc.com:xilinx_board_store:arty:1.1 from remote host https://
github.com/Xilinx/XilinxBoardStore.git
INFO: [Common 17-1573] Object digilentinc.com:xilinx_board_store:arty:1.1
has been installed successfully.
INFO: [Common 17-1570] Installing object
digilentinc.com:xilinx_board_store:arty-a7-100:1.0 from remote host https://
github.com/Xilinx/XilinxBoardStore.git
INFO: [Common 17-1573] Object digilentinc.com:xilinx_board_store:arty-
a7-100:1.0 has been installed successfully.
INFO: [Common 17-1570] Installing object
digilentinc.com:xilinx_board_store:arty-a7-35:1.0 from remote host https://
github.com/Xilinx/XilinxBoardStore.git
```

These boards are downloaded to the default locations which are as follows:

- On Linux - <user home directory>/.Xilinx/Vivado/20xx.x/xhub/board_store/
- On Windows - %APPDATA%\Roaming\Xilinx\20xx.x\xhub\board_store\

Once the boards have been download the repository path to the download location is set automatically and the downloaded boards are now visible in the Default Part page. You can then select the target board and create your project.

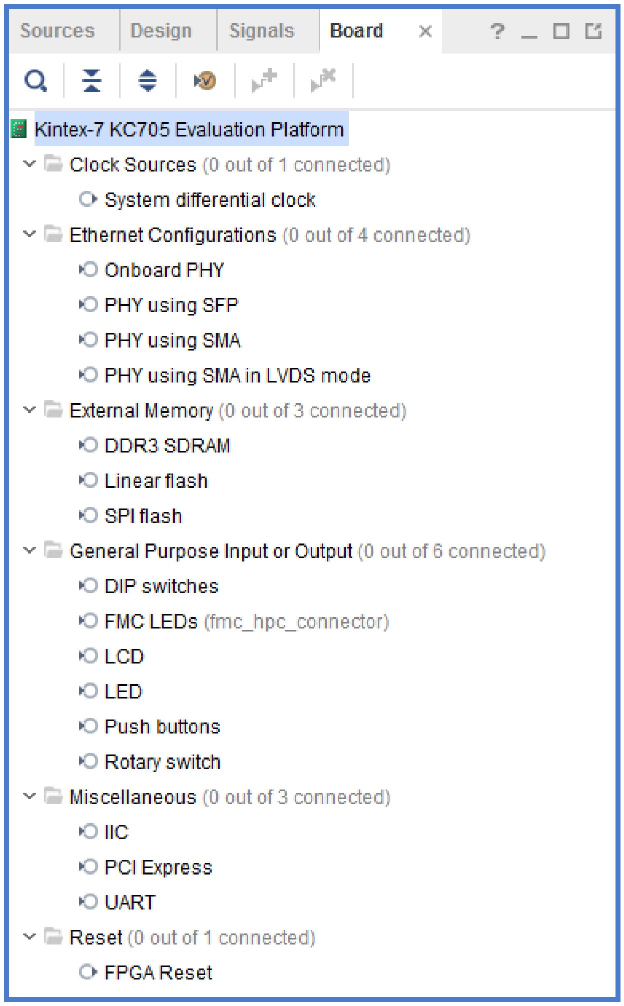
Creating a Block Design to use the Board Flow

The real power of the board flow can be seen in the IP Integrator.

From **Flow Navigator → IP Integrator**, click **Create Block Design** to start a new block design.

As the design canvas opens, you see a Board window, as shown in the following figure.

Figure 165: **Board Window**

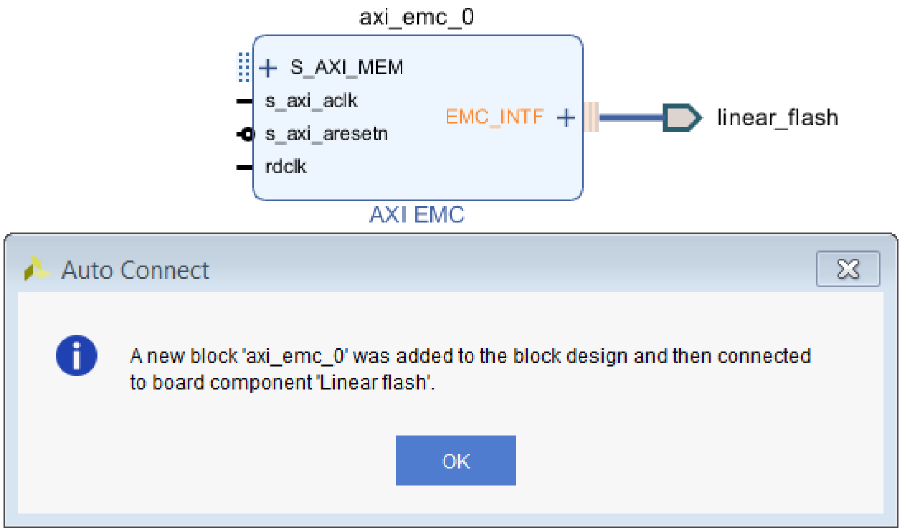


This Board window lists all the possible components for an evaluation board (see the KC705 board above) and a FMC card (if selected). By selecting one of these components, an IP can be quickly instantiated on the block design canvas.

The first way of using the Board window is to select a component from the Board window and drag it onto the block design canvas. This instantiates an IP that can connect to that component and configures it appropriately for the interface in question. It then also connects the interface pin of the IP to an I/O port.

As an example, when you drag and drop the Linear Flash component under the External Memory folder, on the IPI canvas, the AXI EMC IP is instantiated and the interface called `linear_flash` is connected, as shown in the following figure.

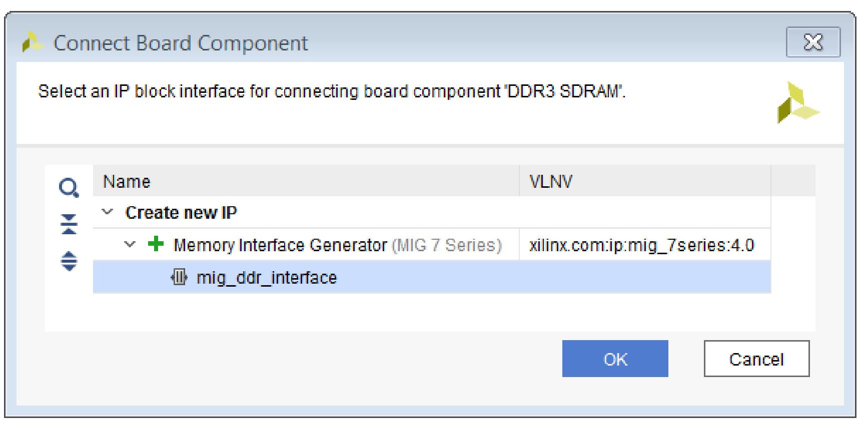
Figure 166: **Dragging and Dropping an Interface on the Block Design Canvas**



The second way to use an interface on the target board is to double-click the unconnected component in question from the Board window.

As an example, when you double-click the DDR3 SDRAM component in the Board window, the Connect Board Component dialog box opens, as shown in the following figure.

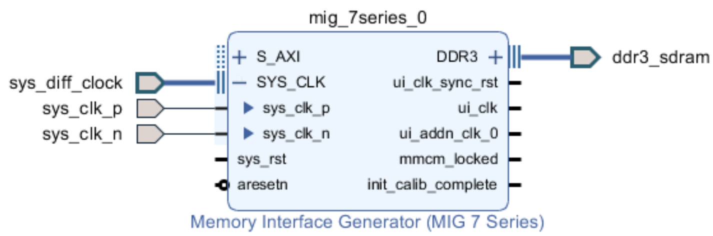
Figure 167: **Connect Board Component Dialog Box**



The `mig_ddr_interface` is selected by default. If there are multiple interfaces listed under the IP, select the interface desired. Select the `mig_ddr_interface`, and click OK.

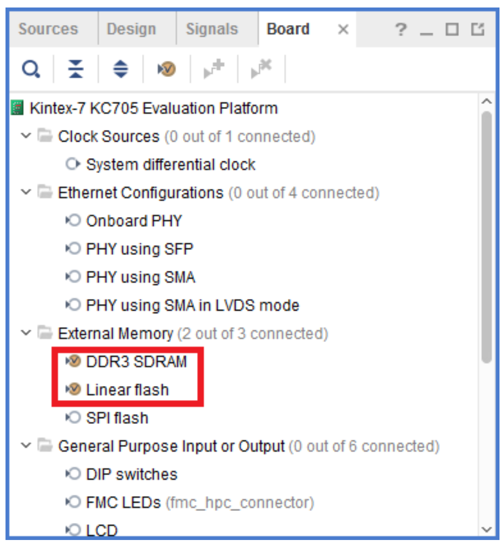
Notice that the IP is placed on the Diagram canvas and connections are made to the interface using the I/O ports. As shown in the following figure, the IP is all configured accordingly to connect to that interface.

Figure 168: **IP instantiated, Configured, and Connected to Interfaces on the Diagram Canvas**



As an interface is connected, that particular interface now shows up as a shaded circle in the Board window, as shown in the following figure.

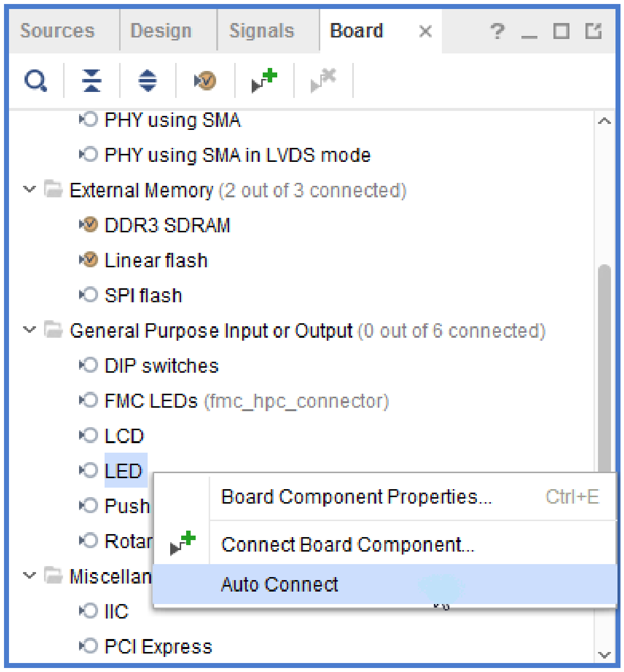
Figure 169: **Board Window after Connecting to an Interface**



A component can also be connected using the Auto Connect command.

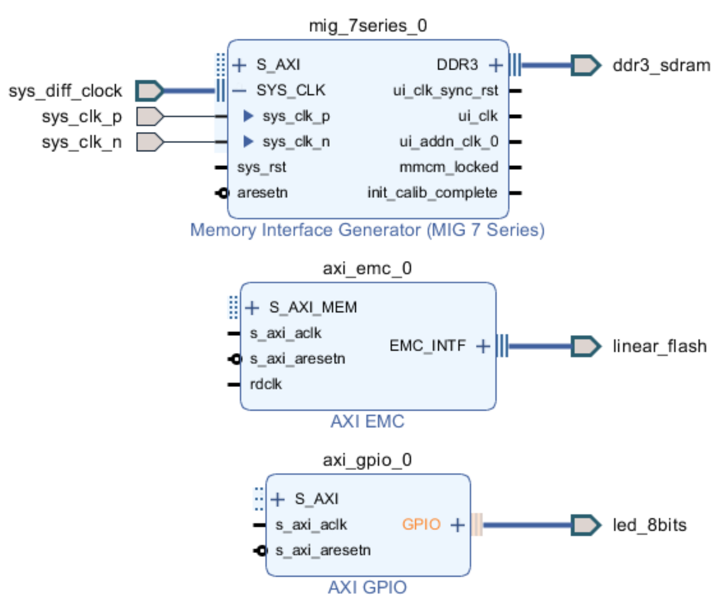
To do this, select and right-click the component and from the menu, as shown in the following figure, and click **Auto Connect**.

Figure 170: **Auto Connect Command**



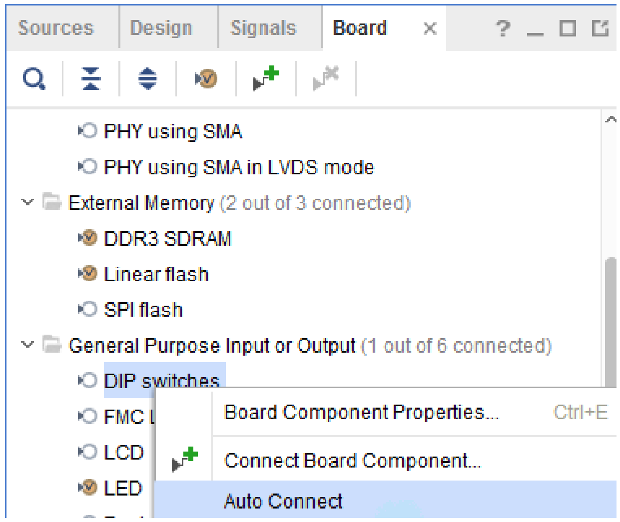
Notice that the GPIO IP has been instantiated and the GPIO interface is connected to the preferred I/O port defined in the Board Interface file, as shown in the following figure.

Figure 171: **Instantiating an IP using Auto Connect**



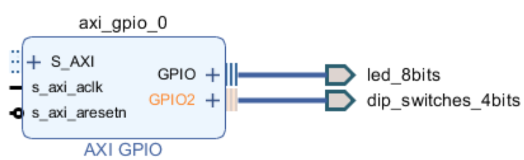
If another component such as `DIP switches` is selected, the board flow is aware enough to know that a GPIO already is instantiated in the design and it re-uses the second channel of the GPIO, shown in the following figure.

Figure 172: **GPIO Auto Connection**



The already instantiated GPIO is re-configured to use the second channel of the GPIO as shown in the following figure.

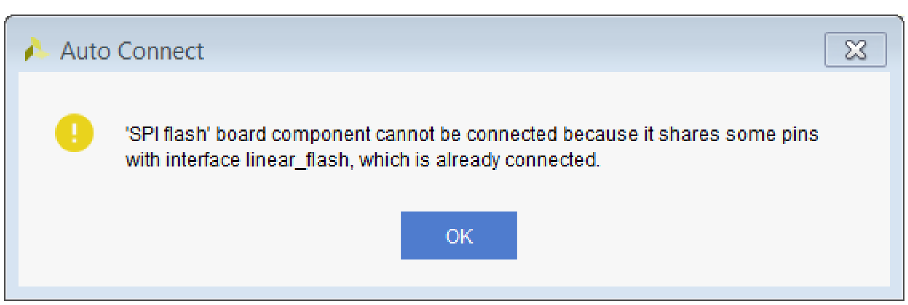
Figure 173: **GPIO IP Configured to Use the Second Channel**



If an external memory component such as the Linear Flash or the SPI Flash is chosen, then as one of them is used, the other component becomes unusable because only one of these interfaces can be used on the target board.

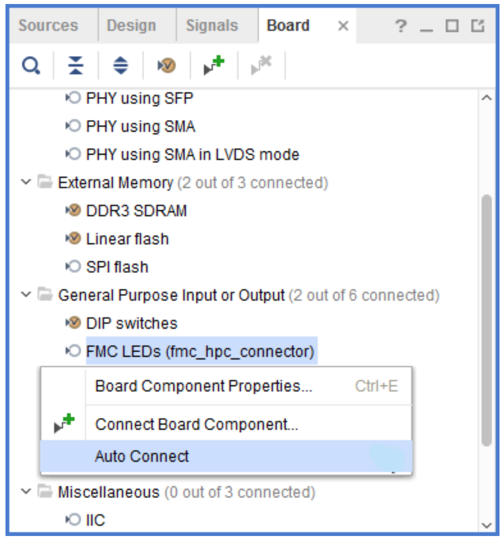
In this case, the following message pops-up when the user tries to drag the other interface such as the SPI Flash on the block design canvas.

Figure 174: **Auto Connect Warning**



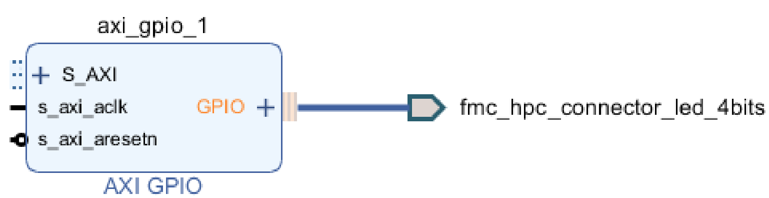
If a component on the FMC card is selected, then that component would be connected using an appropriate IP.

Figure 175: **Connecting to Components on FMC Card**



As can be seen in the following figure, another GPIO has been instantiated that connects to the LEDs on the FMC card.

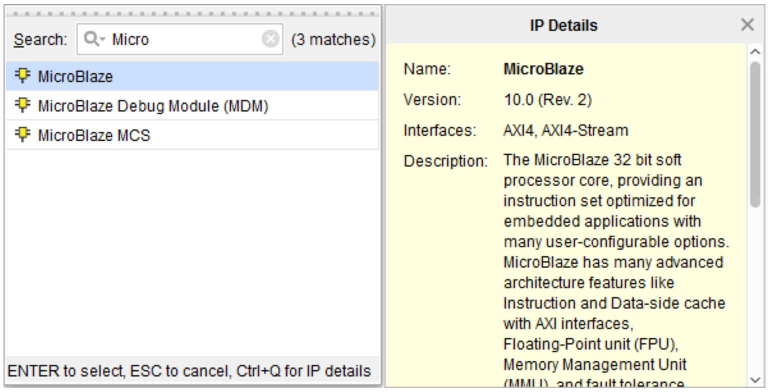
Figure 176: **Connecting to Components on FMC Card**



Completing Connections in the Block Design

After the interfaces you want are in the design, the next step is to instantiate a processor (in case of an processor-based design) or an AXI interconnect if this happens to be a non-embedded design to complete the design.

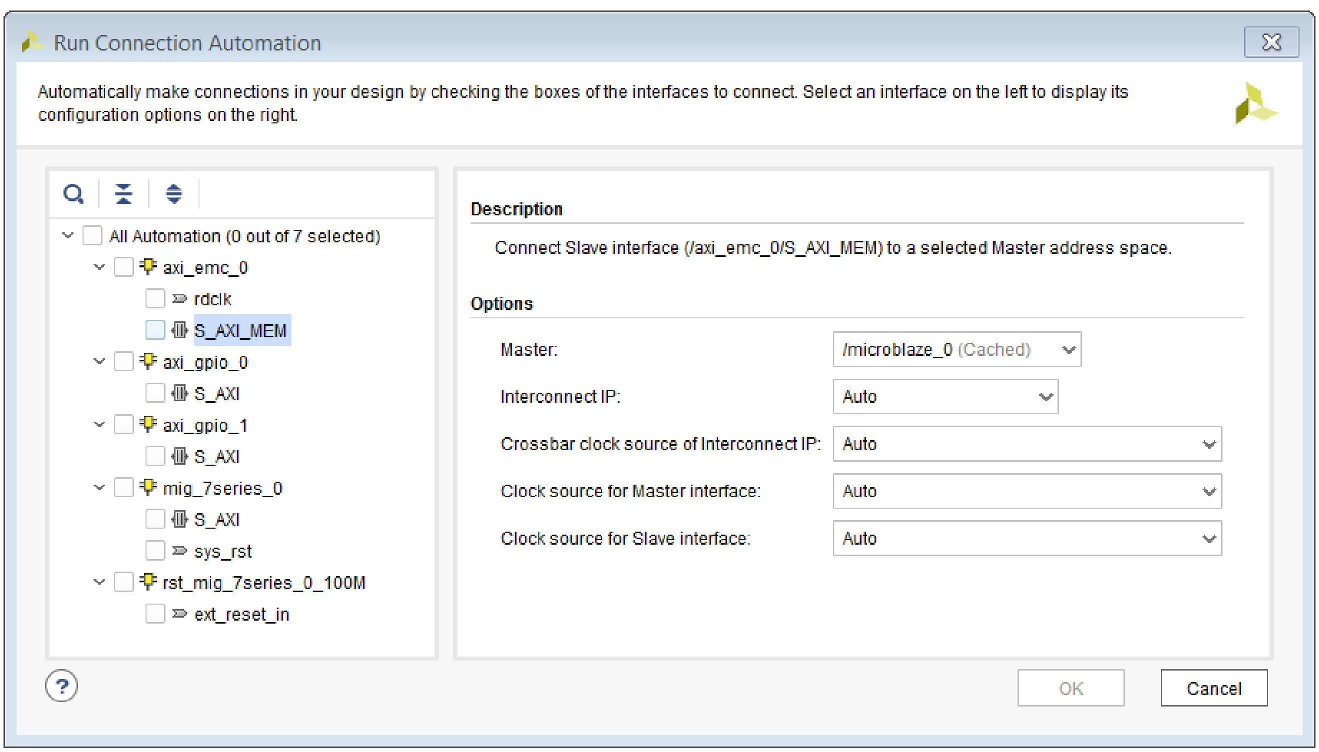
1. To do this, right-click the canvas, and select **Add IP**. From the IP catalog, choose the processor, such as MicroBlaze processor, shown in the following figure.



As the processor is instantiated, Designer Assistance becomes available, as shown in the following figure.

2. Click **Run Block Automation** to configure a basic processor sub-system. The processor sub-system is created which includes commonly used IP in a sub-system such as block memory controllers, block memory generator and a debug module.

   Then you can use the Connection Automation feature to connect the rest of the IP in your design to the MicroBlaze processor by selecting Run Connection Automation. The following figure shows the Run Connection Automation Dialog Box.



The rest of the process is the same as needed for designing in IP Integrator as described in this document.

Archiving a Project when Board Flow is Used

When archiving a project that has board flow enabled, the archive contains a folder which has all the board specific metadata (board files etc). This is specifically useful in cases when a custom board file is used as a part of the board flow. So when the archive is used by another user, there will be no need to add the custom board repository to the project. When the archive is unzipped, a directory is included which contains the board file. This directory is located at:

```
<path to project>/<project_name>/<project_name>.board/<board_name>
```

Chapter 12

Using Third-Party Synthesis Tools in IP Integrator

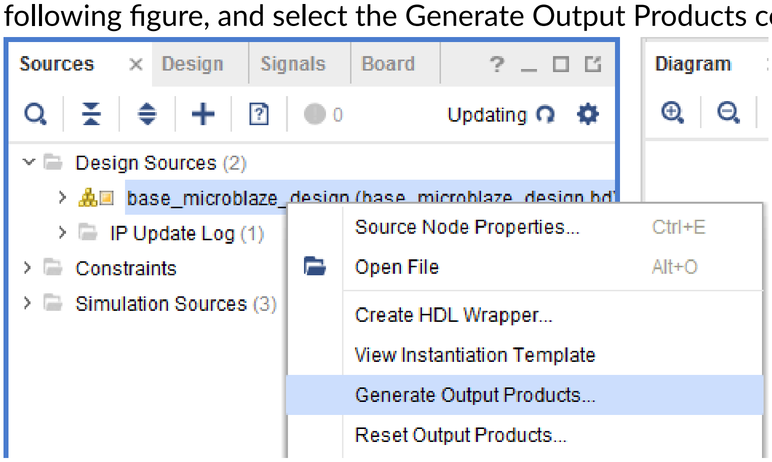
Sometimes it is necessary to use a third-party synthesis tool as a part of the design flow. In this case, you need to incorporate the IP integrator block design as a black box in the top-level design. You can synthesize the top-level of the design in a third-party synthesis tool, write out an HDL or EDIF netlist, and implement the post-synthesis project in the Vivado® environment.

This chapter describes the steps that are required to synthesize the black-box of a block design in a third-party synthesis tool. Although the flow is applicable to any third-party synthesis tool, this chapter describes the Synplify® Pro synthesis tool.

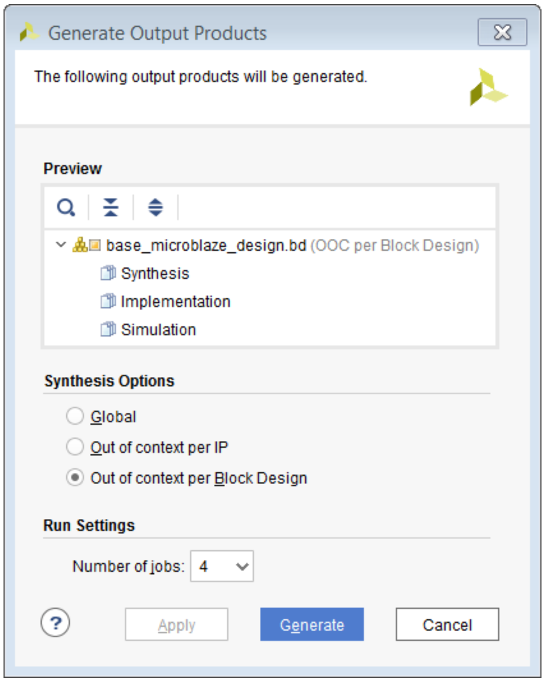
Setting the Block Design as Out-of-Context Module

You can create a design checkpoint (DCP) file for a block design by setting the block design as an Out-of-Context (OOC) module.

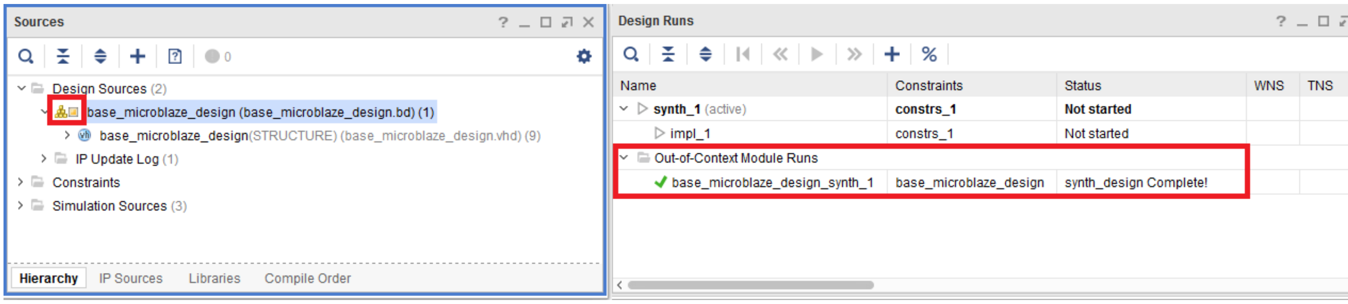
1. Select the block design in the Sources window, right-click to open the menu, shown in the following figure, and select the Generate Output Products command.



2. In the Generate Output Products dialog box, enable the Out-of-Context per Block Design option, as shown below. See Generating Output Products for more information.



A square is placed next to the block design in the Sources window to indicate that the block design has been defined as an out-of-context (OOC) module. The Design Runs window also shows an Out-of-Context Module Run for the block design.



3. When out-of-context synthesis run for the block design is complete, a design checkpoint file (DCP) is created for the block design. The DCP file also shows up in the Sources window, under the block design tree in the IP Sources view. The DCP file is written to the following directory:

```
<project_name>/<project_name>.srcs/<sources_1>/<bd>/<block_design_name>
```

DCPs let you take a snapshot of your design in its current state. The current netlist, constraints, and implementation results are stored in the DCP.

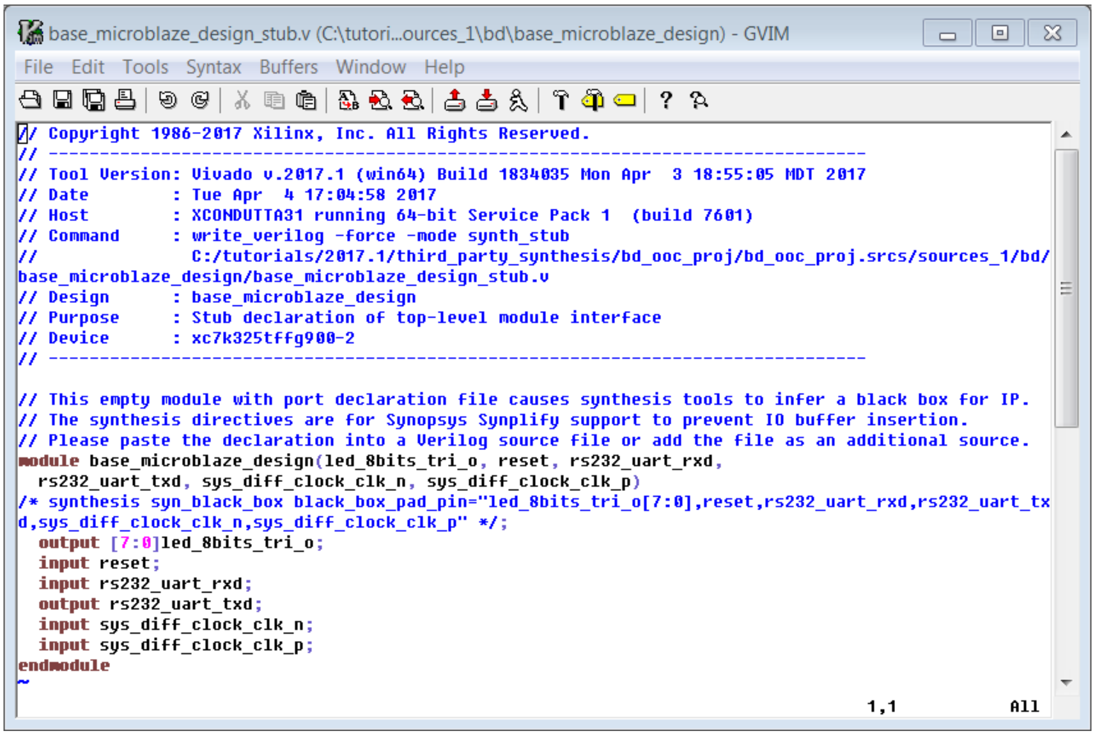
Using DCPs, you can:

- Restore your design if needed
- Perform design analysis

- Define constraints
- Proceed with the design flow

4. When the out-of-context run for the block design is created, two stub files are also created; one each for Verilog and VHDL. The stub file includes the instantiation template which can be copied into the top-level design to instantiate the black box of the block design. An example stub file is shown in the following figure. These files are written to the following directory:

```
<project_name>/<project_name>.srcs/<sources_1>/<bd>/<block_design_name>
```



Creating an HDL or EDIF Netlist in Synplify

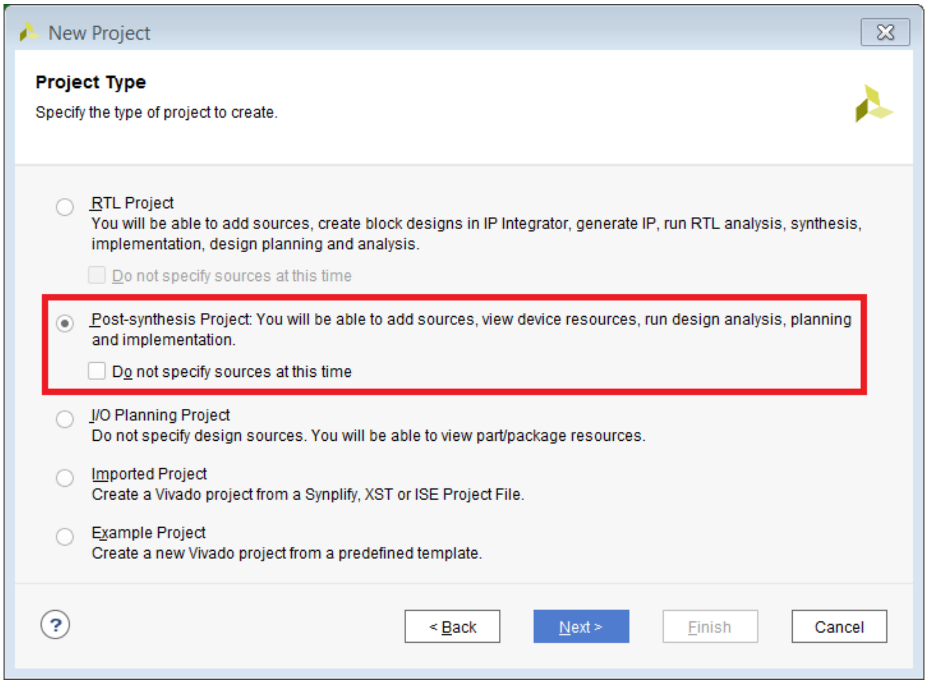
Create a Synplify project and instantiate the black-box stub file (created in Vivado) along with the top-level HDL wrapper for the block design in the Synplify project. The block design is treated as a black-box in Synplify.

After the project is synthesized, an HDL or EDIF netlist for the project can be written out for use in a post-synthesis project.

Creating a Post-Synthesis Project in Vivado

The next step is to create a post-synthesis project in the Vivado IDE. See this link in *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) for more information.

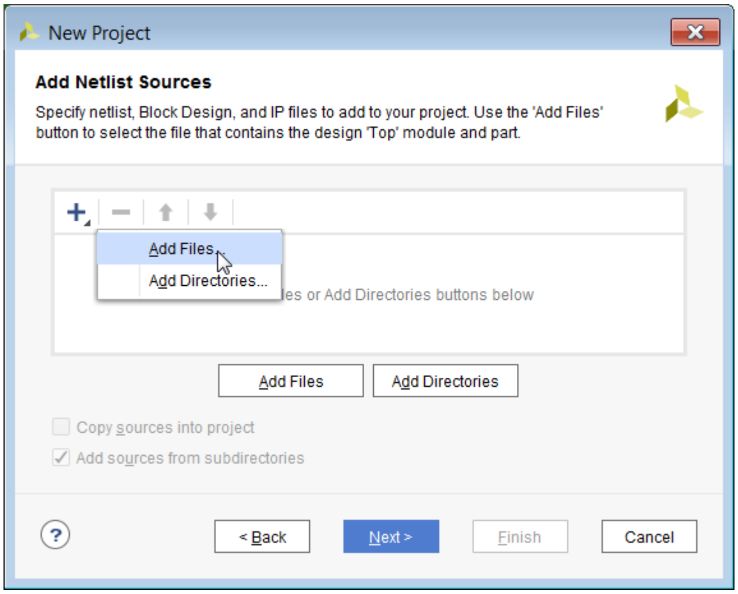
1. Create a new Vivado project, and select **Post-synthesis Project** in the New Project Wizard, as shown in the following figure.



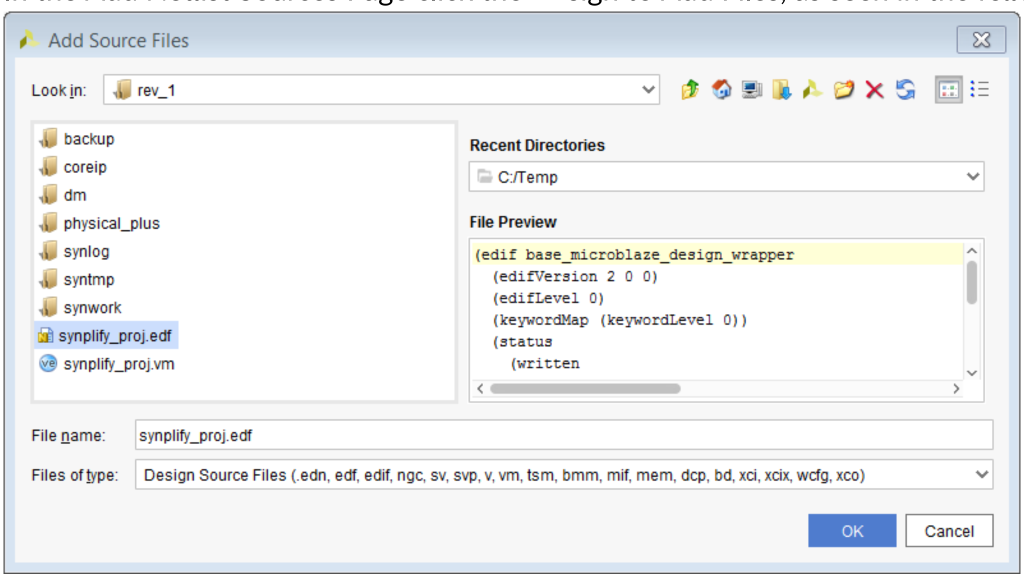
***Note*:** If the Do not specify sources at this time option is enabled, you can add design sources after project creation.

2. Click **Next**.

The Add Sources dialog box opens, as shown in the following figure.

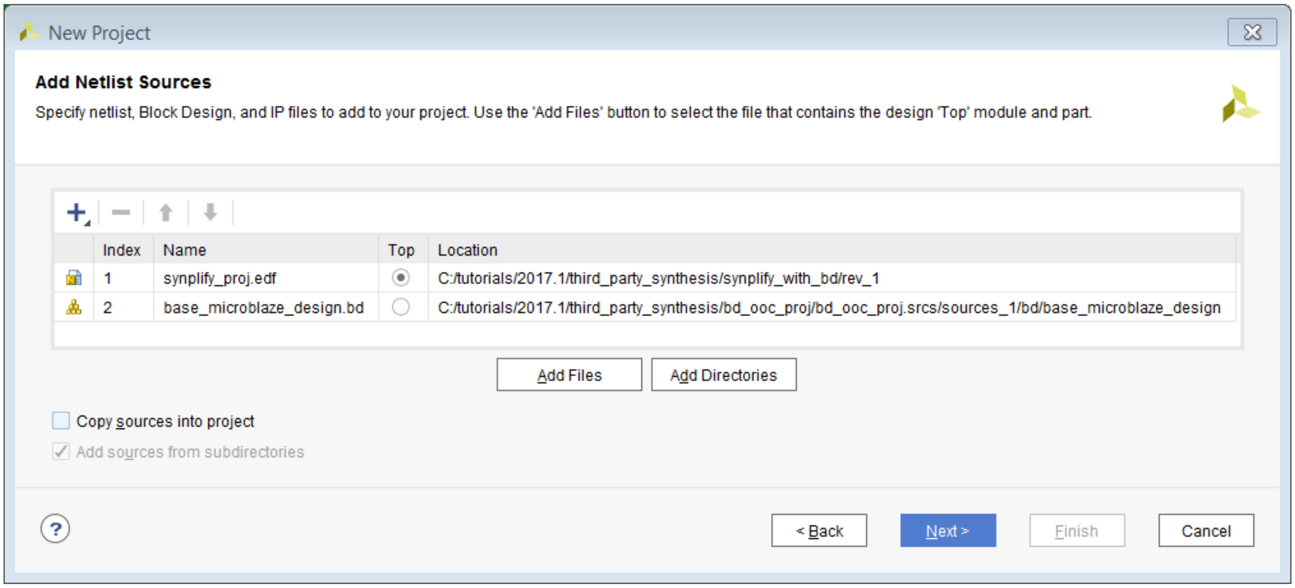


3. In the Add Netlist Sources Page click the '+' sign to Add Files, as seen in the following figure.



4. Select the EDIF netlist for the top-level design, and click **OK**.
5. Using Add Files button or the + sign add the block design file (for which a DCP was created earlier) as well.

   As the block design is added, all the relevant constraints and the DCP file for the block design are picked up by Vivado. The block design is not be re-synthesized. The constraints, however, are reprocessed.



6. Click **Next**.
7. On the Add Constraints page, add any constraints files (XDC) that are needed for the project, and click **Next**.
8. Specify the target part or target platform board as required by the project, and click **Next**.

IMPORTANT! *The target part or platform board for the post-synthesis project must be the same as the project in which the block design was created. If the target parts are different, even within the same device family, the IP used in the block design will be locked, and the design must be re-generated. In that case the behavior of the new block design might not be the same as the original block design.*

9. Verify all the information for the project as presented on the New Project Summary page, and click **Finish**.

   Note: When a block design is added to a netlist project, the block design is "locked". Accordingly, you cannot edit the block design, upgrade it or perform other actions. The block design also needs to be fully generated for it to be a part of a netlist project.

Adding Top-Level Constraints

TIP: *If you did not add the EDIF netlist file, DCP, or design constraints at the time you created the project, you can add those design source files in the current project by right-clicking in the Design Sources window and selecting Add Sources to add files as needed.*

Prior to implementing the design, you must add any necessary design constraints to your project.

The constraints file for the block design are added to the project when you add the block design to the netlist project; however, if you have changed the hierarchy of the block design, then you must modify the constraints in the XDC file to ensure that hierarchical paths used in the constraints have the proper design scope. For more information, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903).

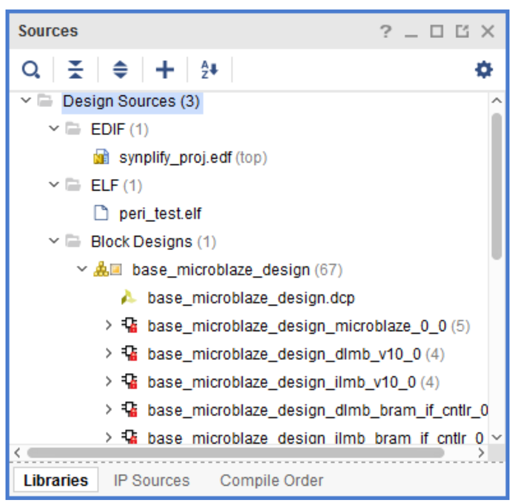
A constraints file can be added to the project at the time it is created, as discussed previously, or tight-click in the Sources window and choose **Add Sources**.

Adding an ELF File

If the block design has an executable and linkable format (ELF) file associated with it, then you will need to add the ELF file to the Vivado project, and associate it with the embedded processor in the block design. See Adding and Associating an ELF File to an Embedded Design for more information on adding the ELF file to the design.

IMPORTANT! *The ELF file must be associated with the netlist project using the SCOPED_TO_REF and SCOPED_TO_CELL properties, and not through the Associate ELF Files command.*

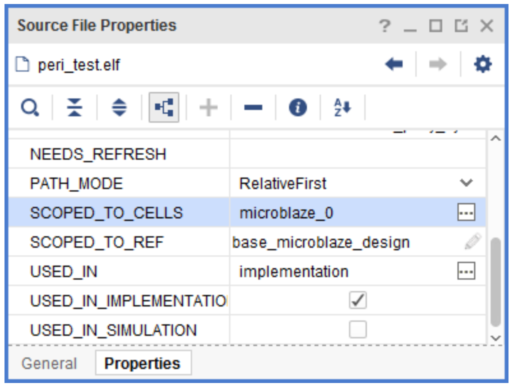
The added ELF file can be seen in the Sources window, as shown above. After the ELF file is added to the project, you must associate the ELF file with the embedded processor design object by setting the SCOPED_TO_REF and SCOPED_TO_CELLS properties.



1. Select the ELF file in the Sources window.
2. In the Source File Properties window, click in the text field of the SCOPED_TO_CELLS and SCOPED_TO_REF properties to edit them.
3. Set the SCOPED_TO_REF property to the name of the block design.

4. Set the SCOPED_TO_CELLS property to the instance name of the embedded processor cell in the block design,

In the following figure, for example, SCOPED_TO_REF is `base_microblaze_design`, and SCOPED_TO_CELLS is `microblaze_0`.



You can also set these properties using the following Tcl commands:

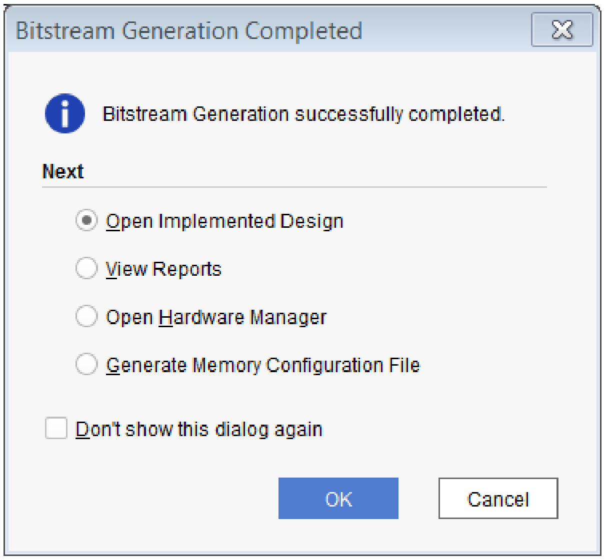
```
set_property SCOPED_TO_REF <block_design_name> [get_files \
<file_path>/file_name.elf]
set_property SCOPED_TO_CELLS { <processor_instance> } [get_files \
<file_path>/file_name.elf]
```

Implementing the Design

Next the design can be implemented and a bitstream generated for the design.

1. In the Flow Navigator, under Program and Debug, click **Run Implementation** or **Generate Bitstream**.

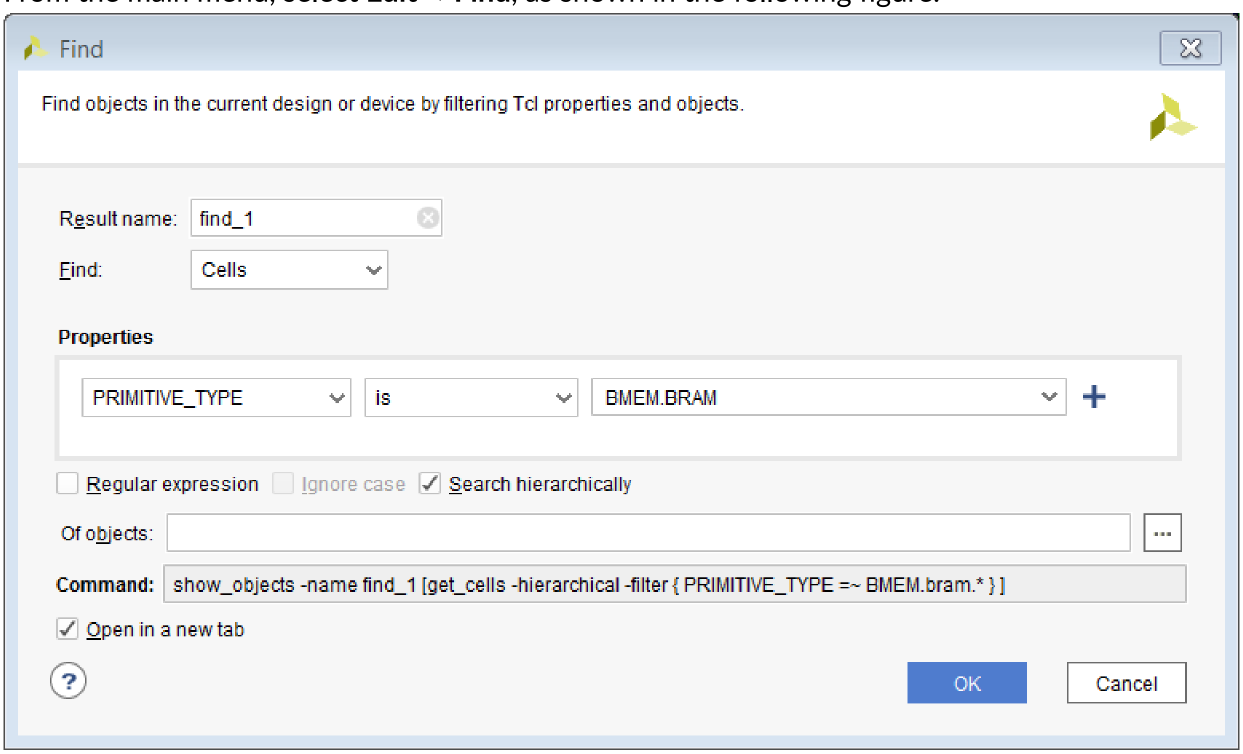
   You are prompted as needed by the Vivado tool to save constraints, and launch implementation, shown in the following figure.



2. In the Bitstream Generation Completed dialog box, click **Open Implemented Design**.

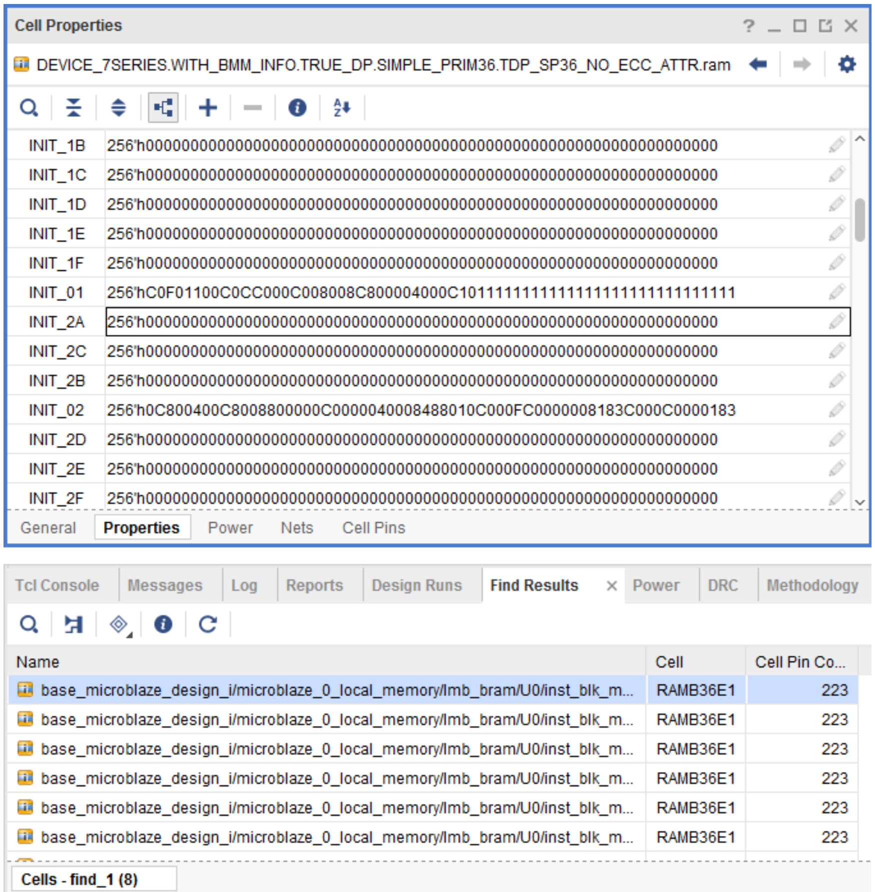
   Verify timing by looking at the Timing Summary report, and ensure that block RAM INIT strings are populated with the ELF data.

3. From the main menu, select **Edit → Find**, as shown in the following figure.



4. In the Find window, set the PRIMITIVE_TYPE to BMEM.BRAM as shown above.
5. Click **OK**.

6. In the Find Results window, select an instance of the block RAM and verify that the INIT properties have been populated in the Cell Properties window, shown in the following figure.



Chapter 13

Referencing RTL Modules

The Module Reference feature of the Vivado® IP Integrator lets you quickly add a module or entity definition from a Verilog or VHDL source file directly into your block design. While this feature does have limitations, it provides a means of quickly adding RTL modules without having to go through the process of packaging the RTL as an IP to be added through the Vivado IP catalog.

Both flows have their benefits and costs:

- The Package IP flow is rigorous and time consuming, but it offers a well-defined IP that can managed through the IP catalog, used in multiple designs, and upgraded as new revisions become available.
- The Module Reference flow is quick, but does not offer the benefits of working through the IP catalog.

The following sections explain the usage of the module reference technology. Differences between the two flows are also pointed in various sections of this chapter.

Referencing a Module

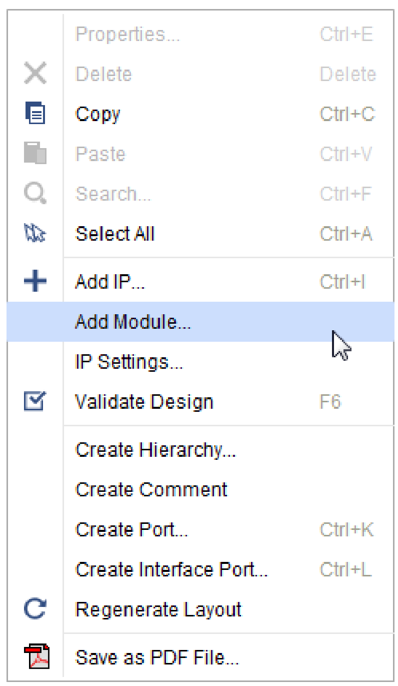
To add HDL to the block design, first you must add the RTL source file to the Vivado project. See this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) for more information on adding design sources. Added source files show up under the Design Sources folder in the Sources window.

An RTL source file can define one or more modules or entities within the file. The Vivado IP Integrator can access any of the modules defined within an added source file, as shown in the following figure.

Figure 177: **RTL Sources in the Sources window**

In the block design, you can add a reference to an RTL module using the Add Module command from the right-click menu of the design canvas, as shown in the following figure.

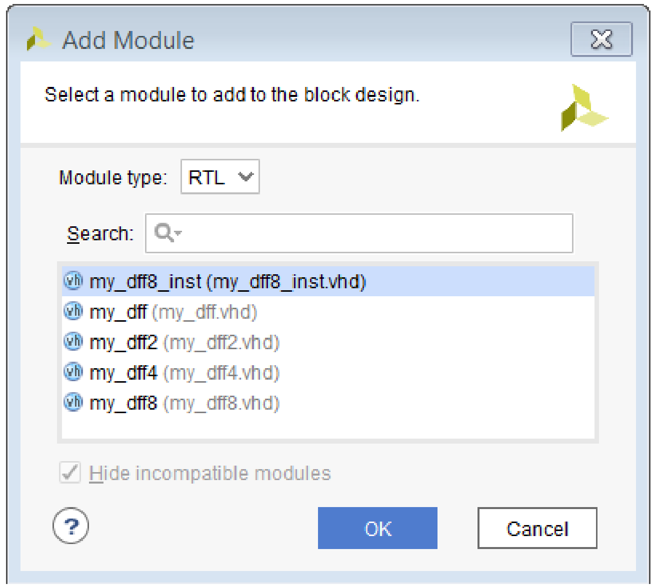
Figure 178: **Add Module Command**



The Add Module dialog box displays a list of all valid modules defined in the RTL source files that you have added to the project. Select a module to add from the list, and click **OK** to add it to the block design, shown in the following figure.

TIP: *You can only select one module from the list.*

Figure 179: **The Add Module Dialog Box**



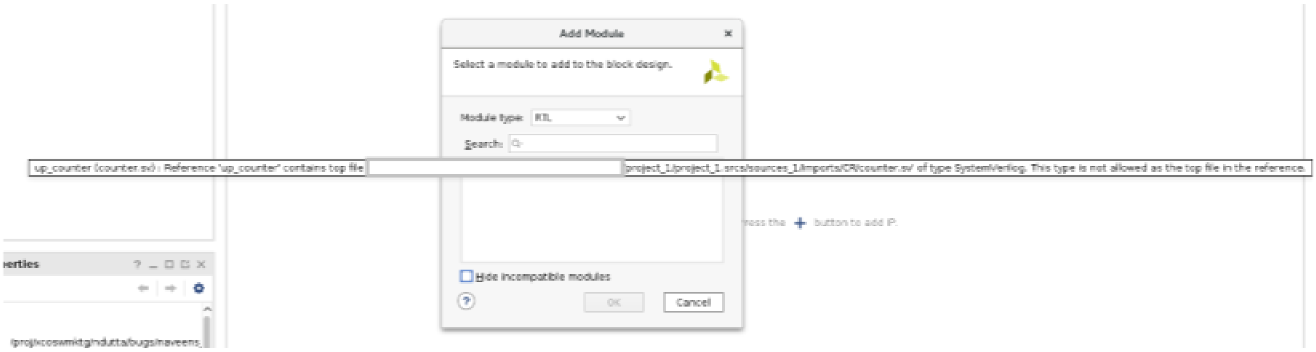
The Add Module dialog box also provides a Hide incompatible modules check box that is enabled by default. This hides module definitions in the loaded source files that do not meet the requirements of the Module Reference feature and, consequently, cannot be added to the block design.

You can uncheck this check box to display all RTL modules defined in the loaded source files, but you will not be able to add all modules to the block design. Examples of modules that you might see when deselecting this option include:

- Files that have syntactical errors
- Modules with missing sources
- Module definitions that contain or refer to an EDIF netlist, a DCP file, another block design, or unsupported IP

Hovering over an incompatible module will show a tool tip with a explanation of why the module is incompatible as shown below.

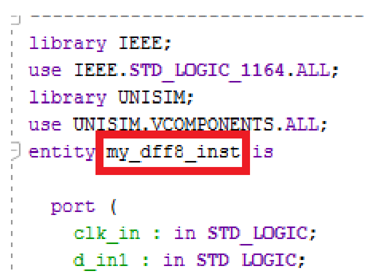
Figure 180: **Incompatible Module Tool-tip**



As shown, in this case the top level file for the module reference is a system verilog file which is not supported by this feature.

The instance names of RTL modules are inferred from the top-level source of the RTL block as defined in the entity/module definition. As shown in the following figure, my_dff8_inst is the top-level entity as shown in the following code sample.

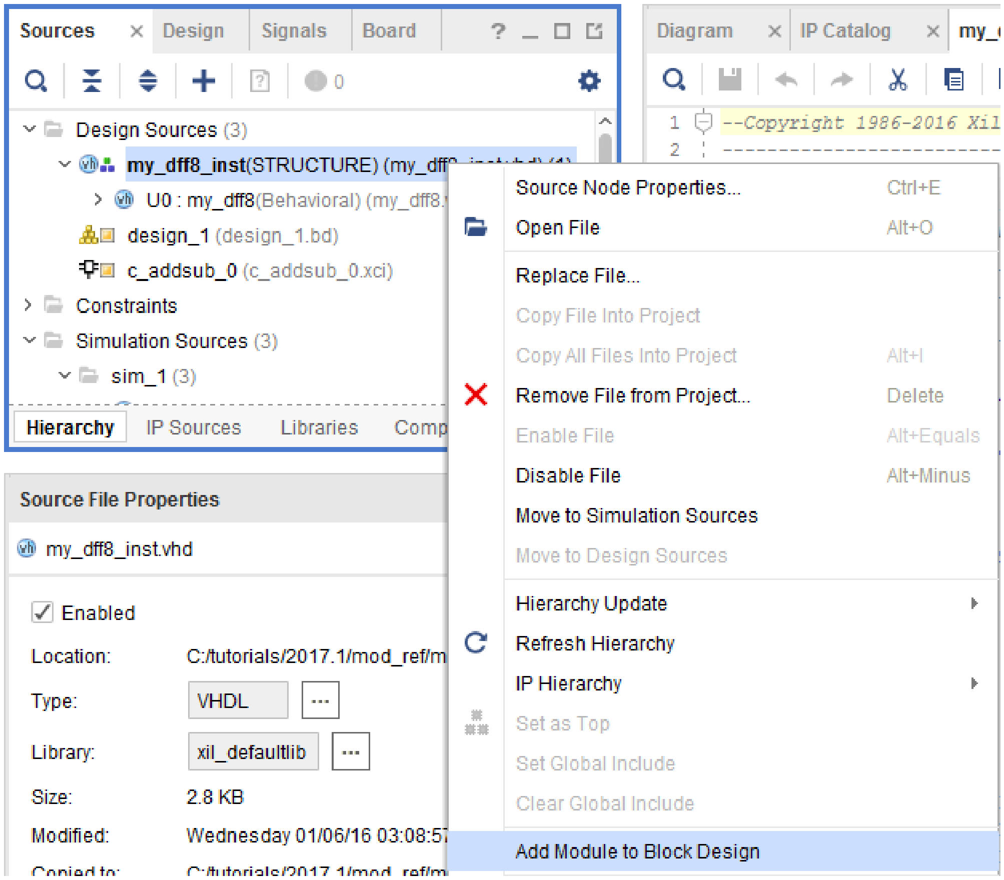
Figure 181: **Inferring Module Names**



IMPORTANT! *If the entity/module name changes in the source RTL file, the referenced module instance must be deleted from the block design and a new module added.*

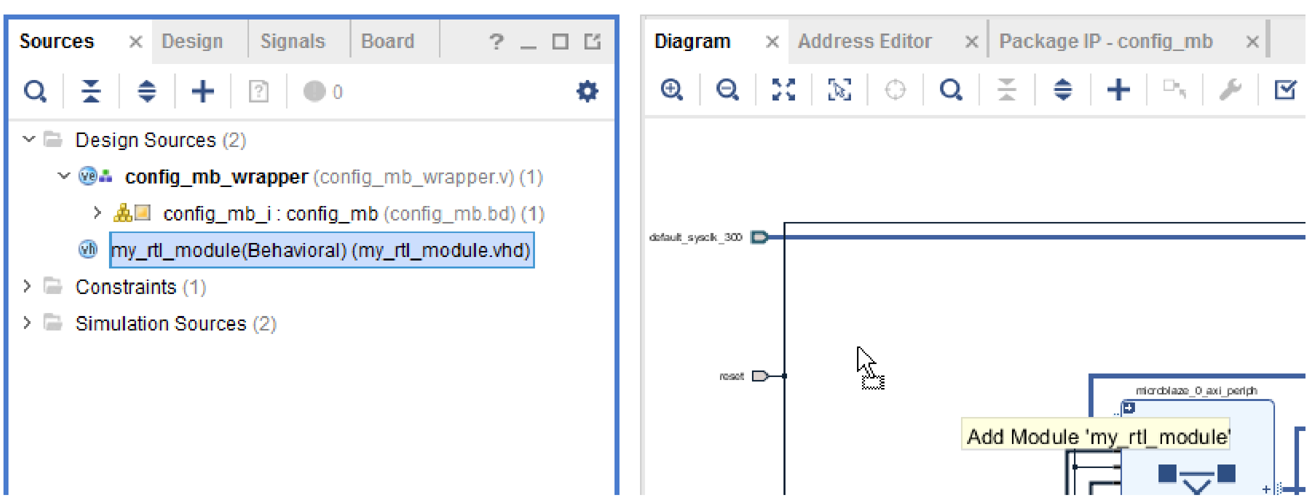
You can also add modules to an open block design by selecting the module in the Sources window and using the Add Module to Block Design command from the context menu, shown in the following figure.

Figure 182: **Alternate Method of Adding Module from Sources Window**



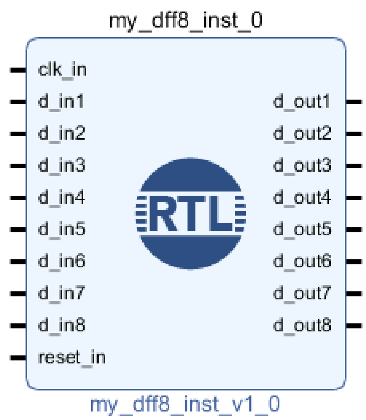
Finally, RTL can also be dragged and dropped from the Sources view onto the block design canvas as shown below.

Figure 183: **Alternate Method of Adding Module from Sources Window**



The IP Integrator adds the selected module to the block design, and you can make connections to it just as you would with any other IP in the design. The IP displays in the block design with special markings that identify it as an RTL referenced module, as shown in the following figure.

Figure 184: **Modules Referenced from RTL Source File**



If a new block design is created after you have added design sources to the project, the block design is not set as the top-level of the design in the Sources window. The Vivado Design Suite automatically assigns a top-level module for the design as the sources are added to the project.

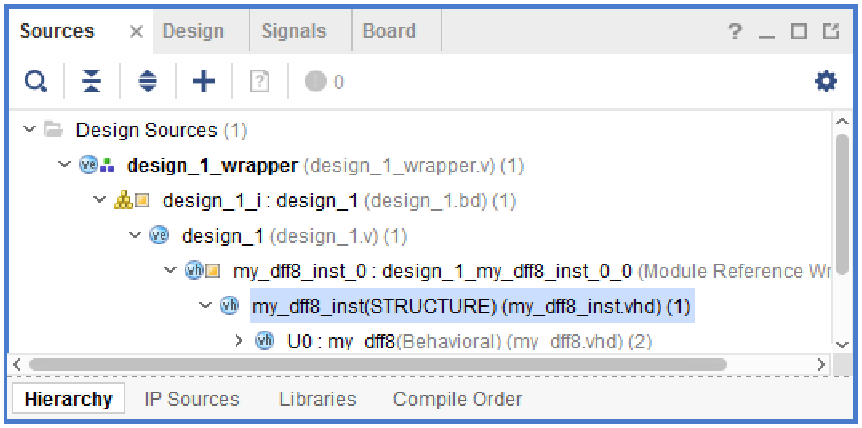
To set the block design as the top level of the design, right-click the block design in the Sources window and use Create HDL Wrapper. See Integrating the Block Design into a Top-Level Design for more information.

TIP: *The block design cannot be directly set as the top level module.*

After creating the wrapper, right-click to select it in the Sources window and use the Set as Top command from the context menu. Any RTL modules that are referenced by the block design are moved into the hierarchy of the design under the HDL wrapper, as shown in the following figure.

If you delete a referenced module from the block design, then the module is moved outside the block design hierarchy in the Sources window.

Figure 185: **Referenced RTL Module under the Block Design Tree**



XCI Inferencing

In some cases, a user code might have commonly-used Xilinx IP instantiated within their RTL. The Reference RTL Module feature allows inferencing the XCI (`.xci`) files for IP embedded within the RTL code.

While a majority of the IP are supported for inferencing, there a few IP that are not supported to be inferenced within the RTL flow. The unsupported IP are, as follows:

- Those with processor data
- Those that support elaboration (HIP)
- Those that support AppCore
- HLS based
    - The `IPDEF` property `ipcomp` includes the string “xilinx_anylanguagehls”
- Do not have RTL enabled
    - The `IPDEF` property `design_tool_contexts` does not include “HDL”
- Hard-coded VLNV list:
    - mig_7series
    - selectio_wiz
    - microblaze_mcs
    - ddr*
    - zynq_ultra_ps_e

- axi_crossbar
- ibert_ultrascale_gth
- ibert_ultrascale_gty
- gtwizard
- microblaze

If an IP from the above list happens to be instantiated within the RTL code, then the Add Module command will fail with the following error:

```
ERROR: [filemgmt 56-181] Reference '<targetName> contains sub-design file
'<xciFile>'. This
sub-design is not allowed in the reference due to following reason(s):
The <vlnv> core does not support module reference.
```

As an example, the code snippet, shown in the following figure, shows that an ILA was instantiated within the RTL code.

Figure 186: **ILA IP Instantiated in RTL**

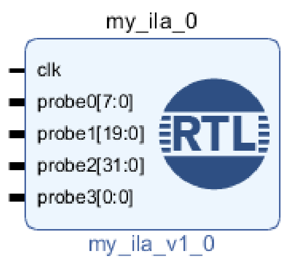
```
module my_ila (
clk,
probe0,
probe1,
probe2,
probe3
);
 input wire clk;
 input wire [7:0] probe0 ;
 input wire [19:0] probe1 ;
 input wire [31:0] probe2 ;
 input wire [0:0] probe3;
ila_0 ila_0_inst (
 .clk(clk), // input wire clk
 .probe0(probe0), // input wire [7:0] probe0
 .probe1(probe1), // input wire [19:0] probe1
 .probe2(probe2), // input wire [31:0] probe2
 .probe3(probe3) // input wire [0:0] probe3
);

endmodule
```

The ILA IP has been configured and added to the project, shown below:

Figure 187: **ILA IP configured and Added to Project**

This RTL can then be added to the block design as an RTL module. It looks like the following figure.

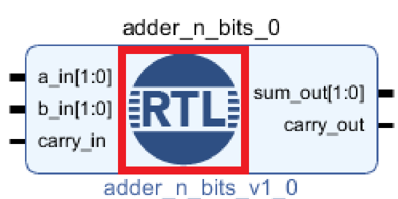
Figure 188: **RTL with ILA IP instantiated as a Module Reference in BD**



IP and Reference Module Differences

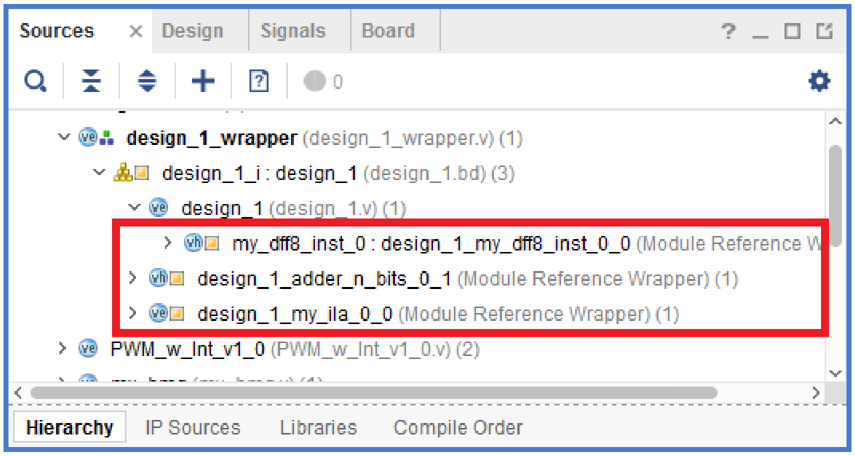
While a referenced module instance looks similar to an IP on the block design canvas, there are some notable differences between an IP and a referenced module. An RTL module in the block design has an "RTL" marking on the component symbol as shown in the following figure.

Figure 189: **RTL Logo on RTL Module Symbol**



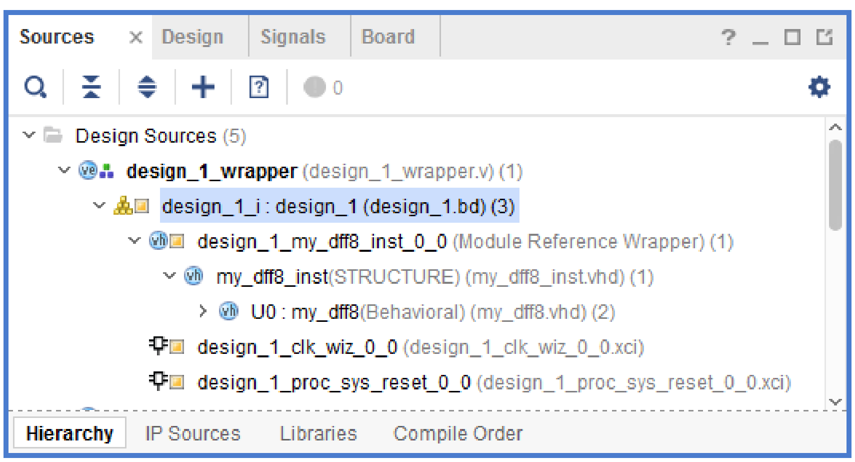
You can also see some differences between packaged IP and referenced modules when viewing the source files in the Sources window. A module reference block shows up in a directory tree with an `_wrapper` extension, and not as an XCI file, as shown in the following figure.

Figure 190: **Top Level of RTL Modules shown as "Module Reference Wrapper"**



When you reset the output products of a block design, the Vivado tools delete the source file, constraint files and other meta data associated with IP blocks; however, a module reference block just contains the source HDL; there is nothing to delete, as shown in the following figure.

Figure 191: **Sources Window After Resetting Output Products**



In this figure, the IP within the project have been reset and there are no HDL under these IP. RTL modules have nothing to reset, so the HDL files show up under the RTL module even after resetting the output products.

Out-of-date IP are shown in the IP Status window, or reported by the appearance of a link in the block design canvas window, as shown in Editing the RTL Module After Instantiation. You can upgrade IP by clicking Upgrade Selected in the IP Status window.

Out-of-date reference modules are also reported by a link in the design canvas window, as shown in Editing the RTL Module After Instantiation. In addition you can force the refresh of a module using the Refresh Module command from the design canvas right-click menu.

While you cannot edit the RTL source files for a packaged IP, you can edit the RTL source for a module reference. Refer to HDL Parameters for Interface Inference for more information.

Because a referenced module is also not a packaged IP, you do not have control over the version of the module instance. The version of a referenced module as displayed in the IP view of the Block Properties window is controlled internally by the Vivado IP Integrator. If you want to have control over the vendor, library, name, and version (VLNV) for a block then you must package the IP as described in the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

For the Module Reference feature there is also no parameter propagation across boundaries. You must use the attributes mentioned in Inferring Control Signals in a RTL Module to support design rule checks run by IP Integrator when validating the design. For example, IP Integrator provides design rule checks for validating the clock frequency between the source clock and the destination. By specifying the correct frequency in the RTL code, you can ensure that your design connectivity is correct.

Inferring Generics/Parameters in an RTL Module

If the source RTL contains generics or parameters, those are inferred at the time the module is added to the block design, and can also be configured in the Re-customize Module Reference dialog box for a selected module.

The following is a code sample for an n-bit full adder, where `adder_width` is the generic that controls the width of the adder.

Figure 192: **Code Snippet for an N-bit Full Adder**

```
entity adder_n_bits is
    generic (adder_width : integer := 2);
    Port ( a_in : in UNSIGNED ((adder_width - 1) downto 0);
           b_in : in UNSIGNED ((adder_width - 1) downto 0);
           carry_in : in STD_LOGIC;
           sum_out : out UNSIGNED ((adder_width - 1) downto 0);
           carry_out : out STD_LOGIC);
end adder_n_bits;
```

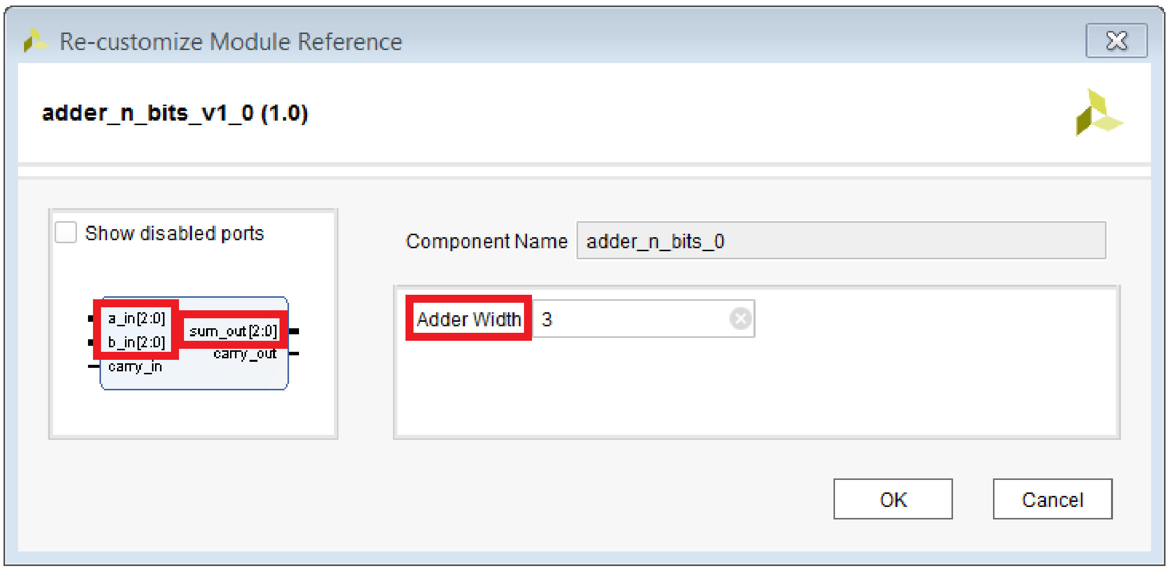
When the adder module is instantiated into the block design, the module is added with port widths defined by the default value for the generic `adder_width`. In this case the port width would be 2-bits.

You can double-click the module to open the Re-customize Module Reference dialog box. You can also right-click the module and select **Customize Block** from the context menu.

Any generics or parameters defined in the RTL source are available to edit and configure as needed for an instance of the module. As the parameter is changed, the module symbol and ports defined by the parameter are changed appropriately.

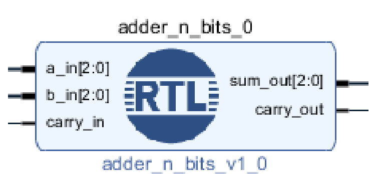
Click **OK** to close the Re-customize Module Reference dialog box and update the module instance in the block design.

Figure 193: **Re-Customize Module Reference Dialog Box**



The symbol in the block design is changed accordingly, as shown below:

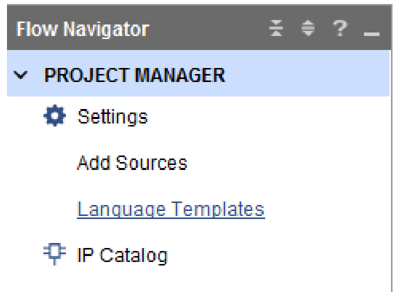
Figure 194: **RTL Module Post-Customization**



Inferring Control Signals in a RTL Module

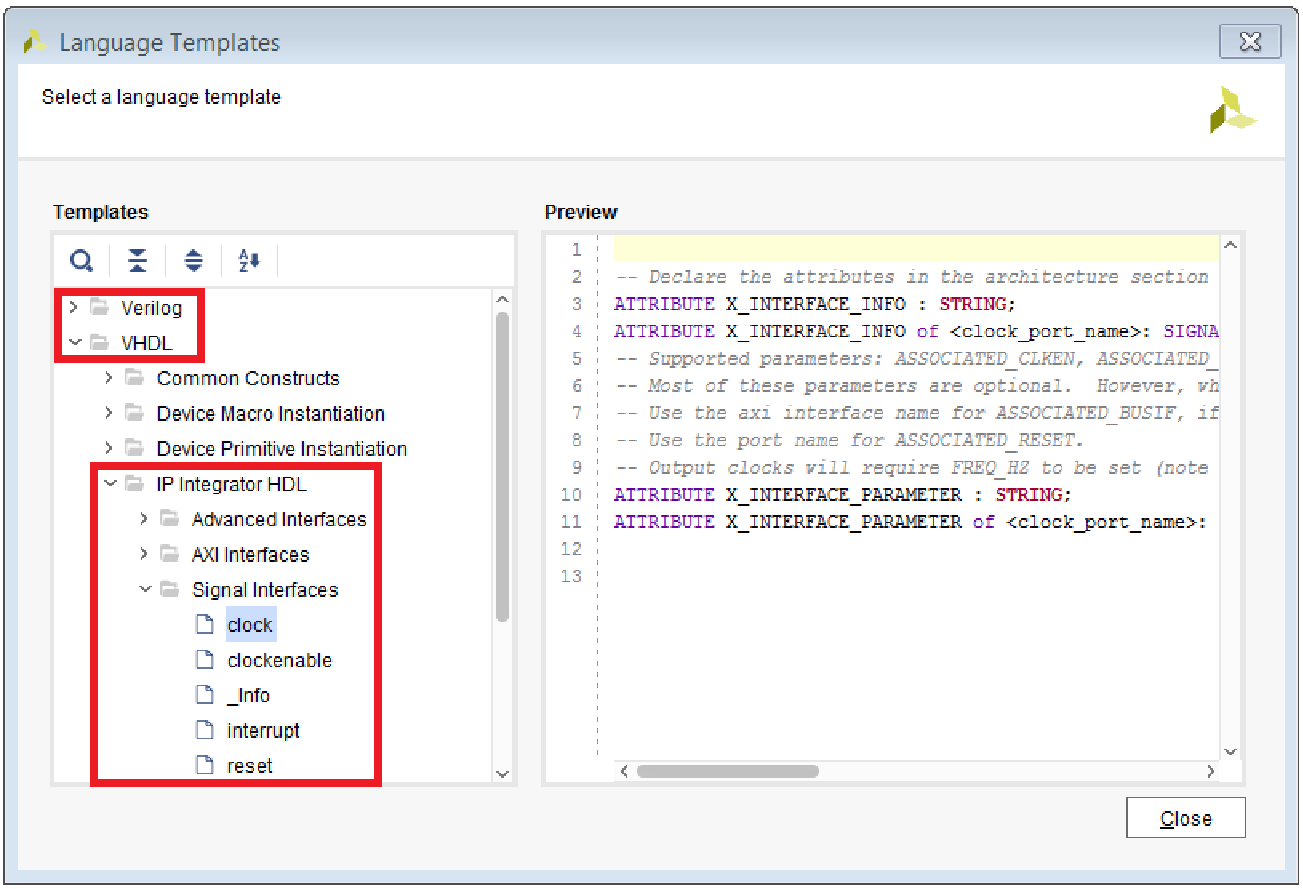
You must insert attributes into the HDL code so that clocks, resets, interrupts, and clock enable are correctly inferred. The Vivado Design Suite provides language templates for these attributes. To access these templates, click **Language Templates** under the Project Manager.

Figure 195: **Select Language Templates**



This opens up the Language Templates dialog box, as shown in the following figure.

Figure 196: **Language Templates Dialog Box**

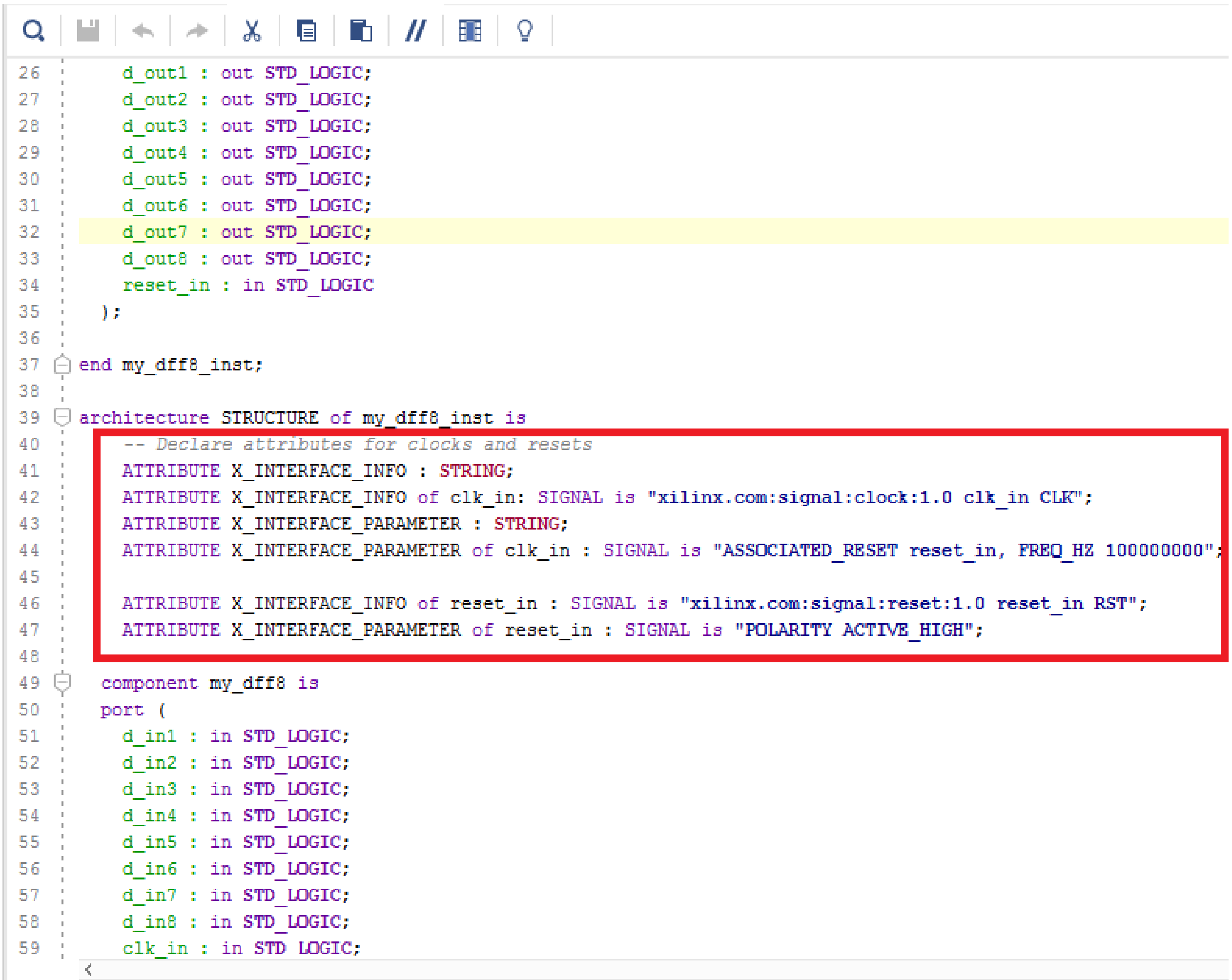


You can expand the appropriate HDL language **Verilog/VHDL → IP Integrator HDL** > and select the appropriate Signal Interface to see the attributes in the Preview pane. As an example, the VHDL language template for the clock interface shows the following attributes that need to be inserted in the module definition.

```
ATTRIBUTE X_INTERFACE_INFO : STRING;
ATTRIBUTE X_INTERFACE_INFO of <clock_port_name>: SIGNAL is
"xilinx.com:signal:clock:1.0 <clock_port_name> CLK";
-- Supported parameters: ASSOCIATED_CLKEN, ASSOCIATED_RESET,
ASSOCIATED_ASYNC_RESET,
ASSOCIATED_BUSIF, CLK_DOMAIN, PHASE, FREQ_HZ
-- Most of these parameters are optional. However, when using AXI, at least
one clock
must be associated to the AXI interface.
-- Use the axi interface name for ASSOCIATED_BUSIF, if there are multiple
interfaces,
separate each name by ':'
-- Use the port name for ASSOCIATED_RESET.
-- Output clocks will require FREQ_HZ to be set (note the value is in HZ
and an
integer is expected).
ATTRIBUTE X_INTERFACE_PARAMETER : STRING;
ATTRIBUTE X_INTERFACE_PARAMETER of <clock_port_name>: SIGNAL is
"ASSOCIATED_BUSIF
<AXI_interface_name>, ASSOCIATED_RESET <reset_port_name>, FREQ_HZ
100000000";
```

Insert these attributes in the HDL code for the module, as shown in the following figure, which shows the declaration of the attributes and the definition of attribute values for both the clock and reset signals.

Figure 197: **Inserting Attributes for Inferring Control Signals**

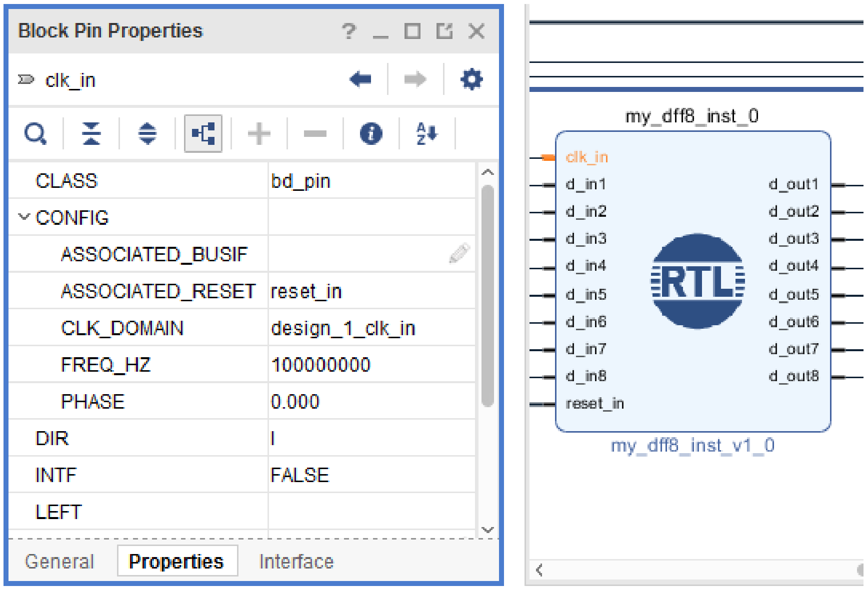


In the code sample shown above, a clock port called clk_in is present in the RTL code. To infer the clk_in port as a clock pin you need to insert the following attributes:

```
-- Declare attributes for clocks and resets
ATTRIBUTE X_INTERFACE_INFO : STRING;
ATTRIBUTE X_INTERFACE_INFO of clk_in: SIGNAL is
"xilinx.com:signal:clock:1.0 clk_in
CLK";
ATTRIBUTE X_INTERFACE_PARAMETER : STRING;
ATTRIBUTE X_INTERFACE_PARAMETER of clk_in : SIGNAL is "ASSOCIATED_RESET
reset_in,
FREQ_HZ 100000000";
```

Notice that the clk_in clock signal is associated with the reset_in reset signal in the attributes shown above. You can click on a pin of a module symbol to see the various associated properties, as shown in the following figure.

Figure 198: **Inspect Inferred Properties of a Clock Pin**



Attributes to infer reset signals are also inserted in the HDL code. Reset signals with names that end with 'n', such as `resetn` and `aresetn`, infer an `ACTIVE_LOW` signal. The tool automatically defines the `POLARITY` parameter on the interface to `ACTIVE_LOW`. This parameter is used in the Vivado IP integrator to determine if the reset is properly connected when the block diagram is generated. For all other reset interfaces, the `POLARITY` parameter is not defined, and is instead determined by the parameter propagation feature of IP integrator. See Chapter 6: Propagating Parameters in IP Integrator, for more information.

TIP: *You can use the X_INTERFACE_PARAMETER attribute to force the polarity of the signal to another value.*

You can also see what IP Integrator has inferred for a referenced module by right-clicking an instance, and selecting **Refresh Module** from the context menu, or by using the following update_module_reference Tcl command:

```
update_module_reference design_1_my_dff8_inst_1_0
```

This reloads the RTL module, and the Tcl Console displays messages indicating what was inferred:

```
INFO: [IP_Flow 19-5107] Inferred bus interface 'clk_in' of definition
'xilinx.com:signal:clock:1.0' (from 'X_INTERFACE_INFO' attribute).
INFO: [IP_Flow 19-4728] Bus Interface 'clk_in': Added interface parameter
'ASSOCIATED_RESET' with value 'reset_in'.
INFO: [IP_Flow 19-4728] Bus Interface 'clk_in': Added interface parameter
'FREQ_HZ'
```

```
with value '100000000'.
INFO: [IP_Flow 19-5107] Inferred bus interface 'reset_in' of definition
'xilinx.com:signal:reset:1.0' (from 'X_INTERFACE_INFO' attribute).
INFO: [IP_Flow 19-4728] Bus Interface 'reset_in': Added interface parameter
'POLARITY' with value 'ACTIVE_HIGH'.
```

This command can also force the RTL module to be updated from the source file. If the source code already contains these attributes prior to instantiating the module in the block design, you see what is being inferred on the Tcl console.

You might want to disable automatic port inferencing. For such cases, you can use the X_INTERFACE_IGNORE attribute. The syntax for VHDL is as follows:

```
ATTRIBUTE X_INTERFACE_IGNORE:STRING;
ATTRIBUTE X_INTERFACE_IGNORE OF <port_name>: SIGNAL IS "TRUE";
```

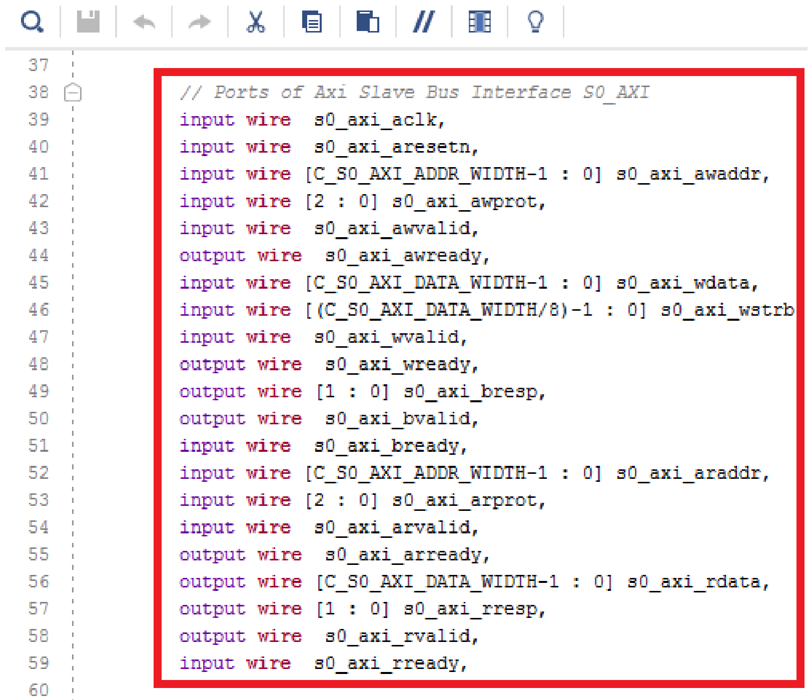
The syntax for Verilog is as follows:

```
(* X_INTERFACE_IGNORE = "true" *)
input <port_name>,
```

Inferring AXI Interfaces

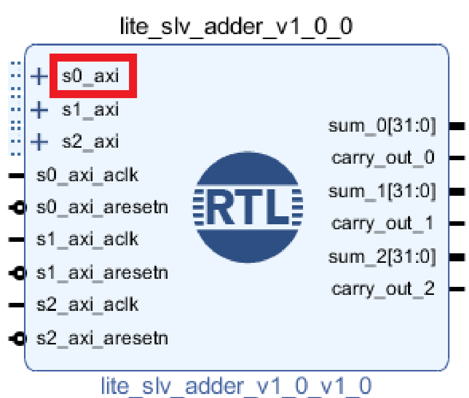
When you use the standard naming convention for an AXI interface (*recommended*), the Vivado IP Integrator automatically infers the interface. As an example, the following code sample shows standard AXI names being used.

Figure 199: **Inferring AXI Interface when standard naming convention is used**



When this RTL module is added to the block design the AXI interface is automatically inferred as shown below.

Figure 200: **AXI Interface Inferred on Module Reference**

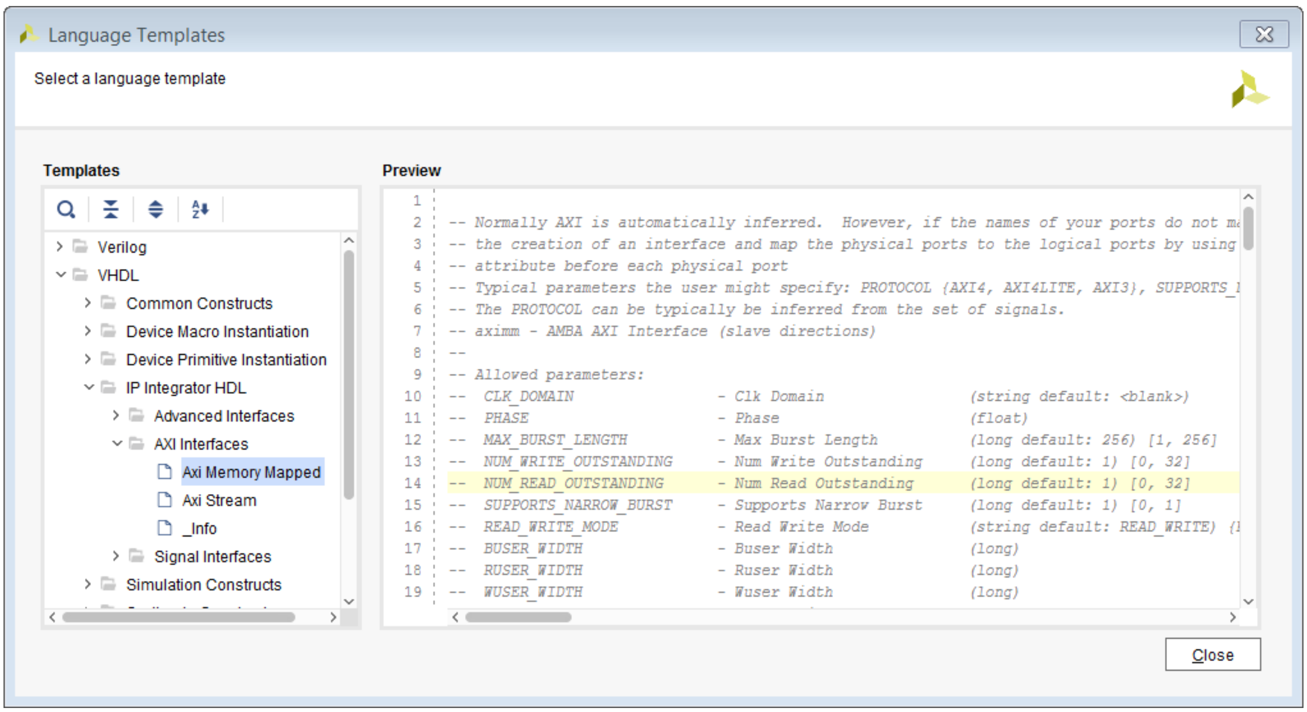


After an AXI interface is inferred for a module, the Connection Automation feature of IP Integrator becomes available for the module. This feature offers connectivity options to connect a slave interface to a master interface, or the master to the slave.

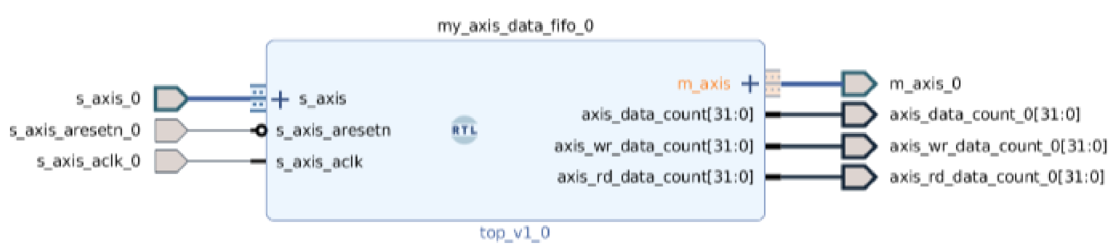
If the names of your ports do not match with standard AXI interface names, you can force the creation of an interface and map the physical ports to the logical ports by using the X_INTERFACE_INFO attribute as found in the Language Templates.

Expand the appropriate HDL language Verilog/VHDL > IP Integrator HDL and select the appropriate AXI Interface to see the attributes in the Preview pane. As an example, the following figure shows the VHDL language template for the AXI4 interface listing the attributes that need to be inserted into the module definition.

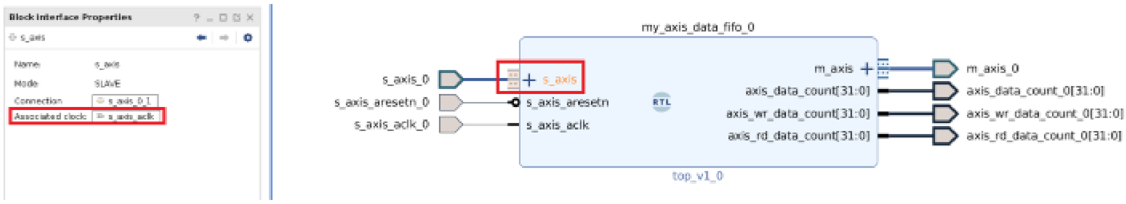
Figure 201: **Use Attributes Specified in the Language Templates for Non-Standard AXI Names**



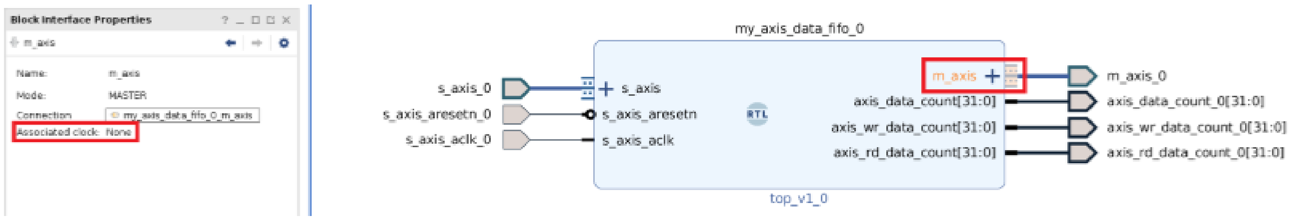
If the same axi clock is to be associated with a slave as well as a master interface, the clock should be called **axi_aclk** or **axis_aclk** instead of calling the clock **s_axis_aclk** or **m_axis_aclk**. Keeping the prefix "m_" and "s_" out from the clock name infers that the clock is to be associated with both master and slave axi interfaces. As an example in the following figure an IP is shown with a AXI streaming slave and an AXI streaming master interface.



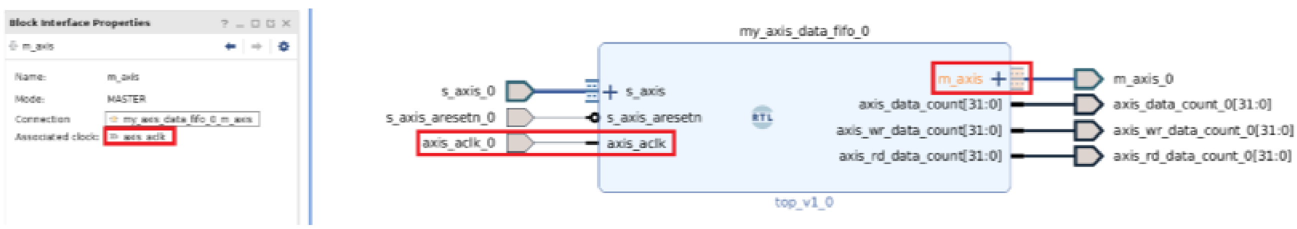
If you look at the properties window for the slave interface, **s_axis**, it shows that the associated clock is **s_axis_aclk**.



If you look at the properties of the **m_axis** interface, the Associated clock value is set to **None**.



To auto associate the **s_axis_aclk** to both the **s_axis** and the **m_axis** interfaces, rename the clock in your RTL code to **axis_aclk**.



As you can see now the clock is associated to the **m_axis** interface as well.

Prioritizing Interfaces for Automatic Inference

In some cases users may need to specify the order in which interfaces are inferred rather than letting the tools automatically infer them. The Module Reference feature allows the user to prioritize the order of the interface inference. There are several attributes that can be used to infer interfaces.

For a particular interface, you might have slightly different physical pin (port) names than that prescribed in the standard. In such cases, specify the following attribute on the Tcl command line:

```
(* X_INTERFACE_INFO = "xilinx.com:interface:axis:1.0 axi_stream_s2c TREADY"
*)
output axi_stream_s2c_tready,
```

This attribute is inserted above the port definition in the HDL code, and specifies that the interface to be inferred has a VLNV of `xilinx.com:interface:axis:1.0`, its name is `axi_stream_s2C`, with a logical pin name `TREADY` to be mapped to the physical pin name `axi_stream_s2c_tready`. This attribute has the highest priority than other inferencing attributes.

If you have multiple versions of an interface that are slightly different in behavior or ports, use the `X_INTERFACE_PRIORITY_LIST` attribute to infer one over the other. The Verilog syntax for this is, as follows:

```
(* X_INTERFACE_PRIORITY_LIST = "xilinx.com:dsv:dsv_axis:3.0" *)
module axi_stream_gen_check #(
 ....
 ....
)
```

The VHDL syntax is, as follows:

```
entity HDMI_TX_INTF is
 Port (
-- put ports here
);

attribute X_INTERFACE_PRIORITY_LIST : string;
attribute X_INTERFACE_PRIORITY_LIST of HDMI_TX_INTF : entity is
"xilinx.com:user:my_hdmi:3.0 xilinx.com:cust:cust_hdmi:4.0";
end HDMI_TX_INTF;
```

This attribute infers the specified interface as opposed to any other similar types of interfaces in the repository. This attribute needs to be inserted before the module definition in Verilog, and in the entity body in VHDL. This attribute has the second highest priority.

Interface inferencing can also be done by adding properties in the project as shown in the following code snippet:

```
set_property ip_interface_inference_priority xilinx.com:user:my_axis:2.0
[current_project]
```

This has the third highest priority.

Finally, the repository ordering in the settings of the project determines the order of inferencing. As can be seen in the following figure, there are two repositories containing custom interfaces added to the project. The repository specified at the top:

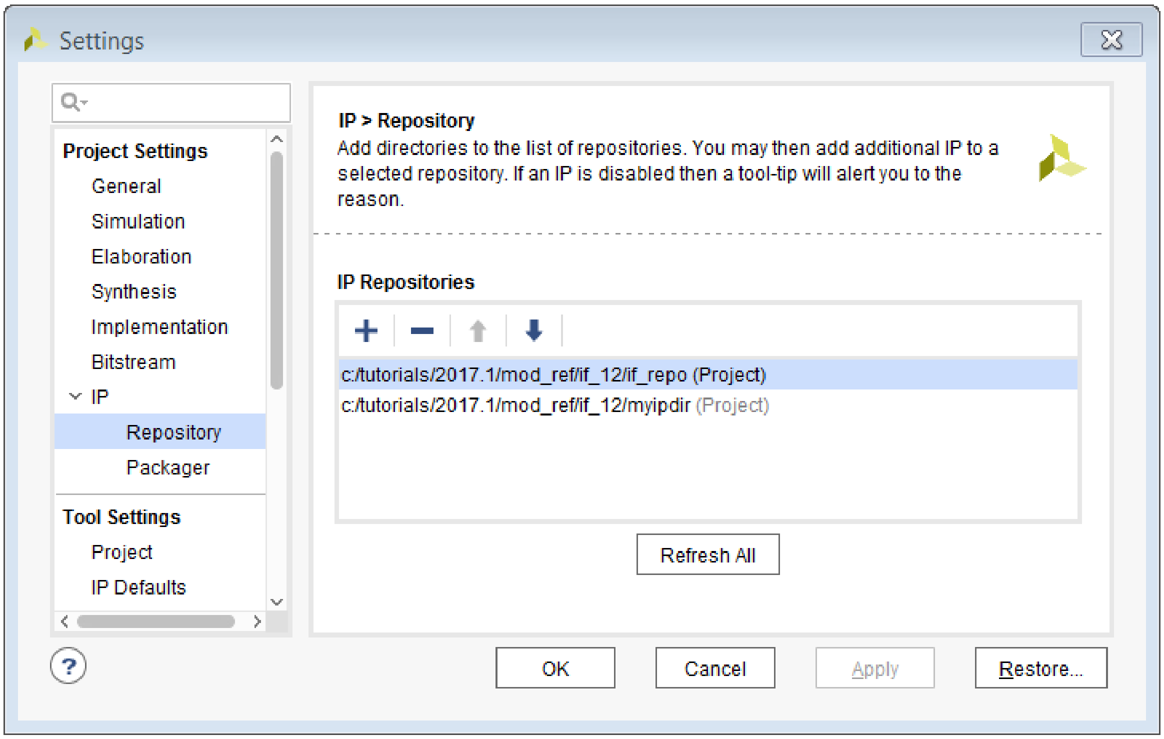
```
C:/tutorials/2018.2/if_12/if_repo
```

takes precedence over

```
C:/tutorials/2018.2/mod_ref/if_12/myipdir.
```

Typically, if you follow the naming conventions, then just adding the repositories in the project should be sufficient to infer an interface. See the following figure.

Figure 202: **Add Repositories Containing Interfaces, Based on Priority**



HDL Parameters for Interface Inference

The IP Packager and the Module Reference flow support a number of Attributes of the style `X_[...]` that can specify a certain behavior to replace and modify the standard interface inference heuristic. As a global rule, the parameters always take precedence over any project-wide or application-wide behavior. Furthermore, most attributes are attached to the ports (because VHDL or Verilog do not have any notion of an interface that this information could be attached to). If the attribute relates to interface-wide information (for example, X_INTERFACE_MODE), the attribute applies to the entire interface, and any constituent port can be chosen as representative for the whole interface.

General Usage

VHDL

Add the attribute to the architecture section as shown below.

```
architecture arch_impl of my_module is
ATTRIBUTE X_INTERFACE_INFO : STRING;
ATTRIBUTE X_INTERFACE_INFO of s_tready: SIGNAL is
"xilinx.com:interface:axis:1.0
s_axi TREADY";
```

Verilog

Prefix the comment to the affected construct as shown below.

```
(* X_INTERFACE_INFO = "xilinx.com:interface:axis:1.0 s_axi TREADY" *)
output s_tready,
```

List of Supported X_ Attributes

The following is a list of all attributes supported by the IP packager, Module Reference flow, and their components.

X_INTERFACE_INFO

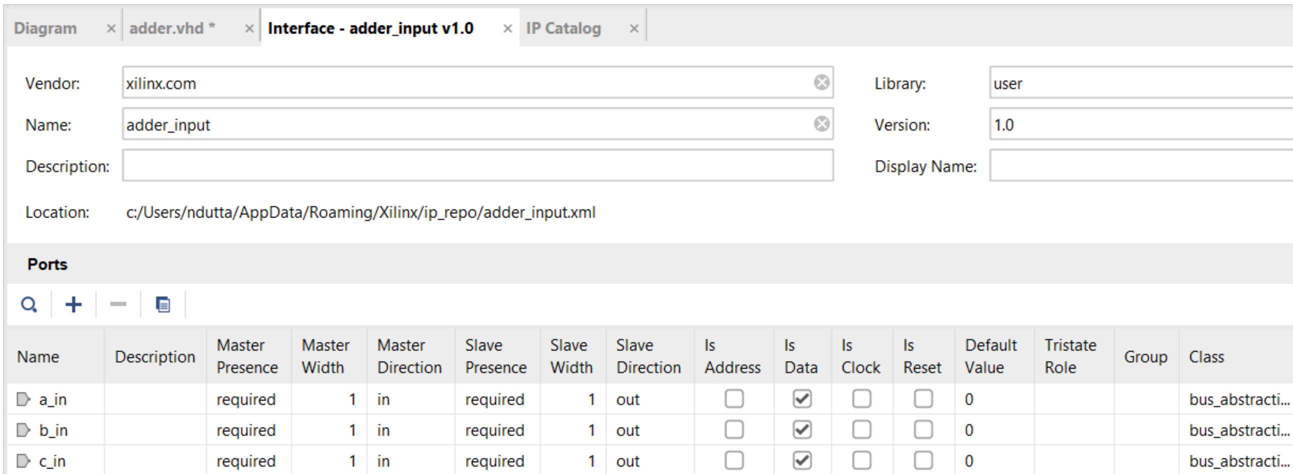
- Attach to: Port
- Syntax: `VLNV INTERFACE_NAME LOGICAL_NAME[,VLNV INTERFACE_NAME LOGICAL_NAME etc]`
- or `VLNV`

The first variant creates an interface according to the bus definition specified by `VLNV`, with the name `INTERFACE_NAME` and maps the port attached to the logical port `LOGICAL_NAME`. Note that this needs to be specified for every port that must be part of the created interface, because the heuristic will not add any ports to this user created interface automatically. Through the addition of multiple triplets here, a port can be added to multiple interfaces, if desired.

The second variant only specifies the `VLNV` of the interface that this port will be a part of. Vivado takes care of adding the individual ports, and inferring a name and the logical-to-physical mapping.

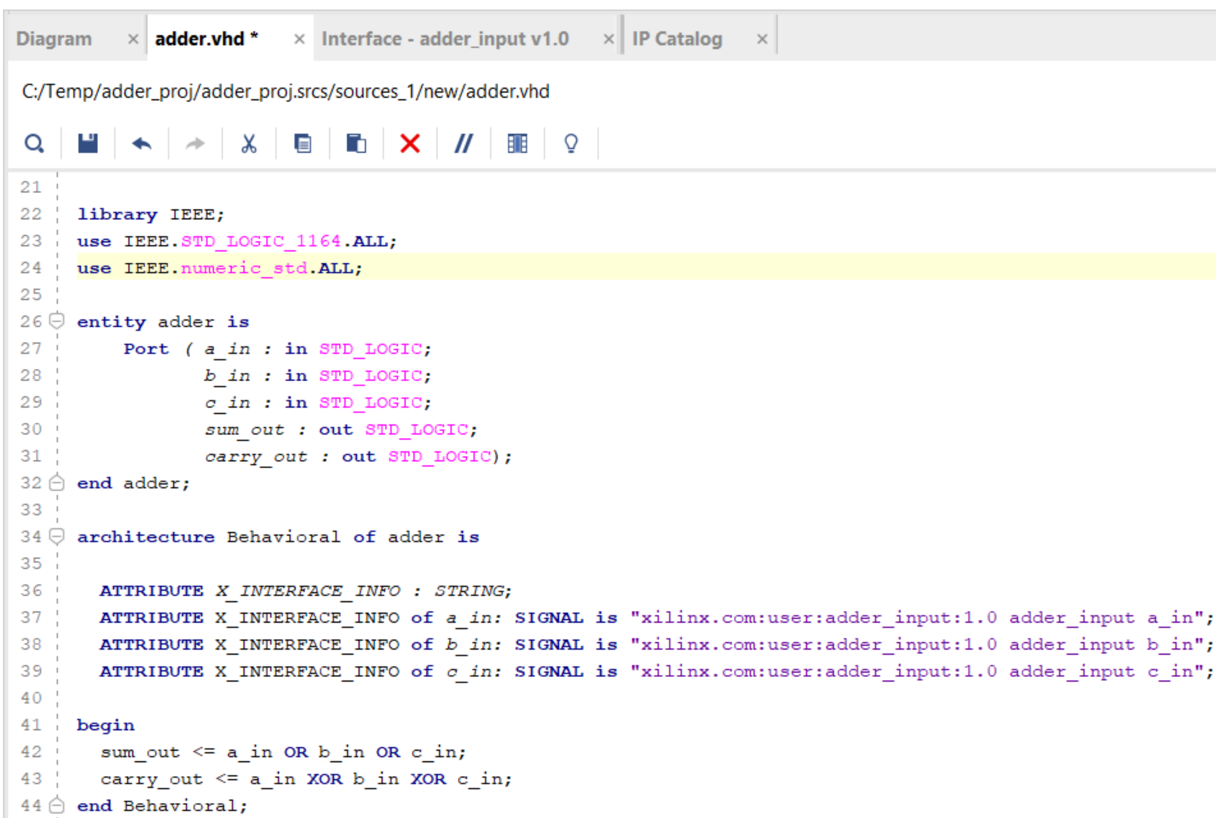
As an example, the code snippet in Figure 204: Adding Pins or I/O ports as a part of an Interface Port shows how ports can be shown as being a part of the interface called `adder_input`. The `adder_input` interface exists in the IP catalog with all the ports correctly specified as shown in the following figure.

Figure 203: **Pre-existing interface in Repository**



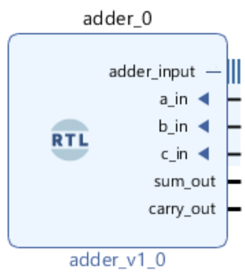
Given the pre-existing interface, attributes can be inserted in the VHDL source code below to make the ports of the module a part of the interface.

Figure 204: **Adding Pins or I/O ports as a part of an Interface Port**



When the RTL code above is instantiated on the block design as a module reference block, the block design looks as follows.

Figure 205: **Module Reference highlighting X_INTERFACE_INFO attribute**



X_INTERFACE_PARAMETER

- Attach to: Port
- Syntax: `NAME VALUE [,NAME VALUE etc]`
- or `"XIL_INTERFACENAME" IFC_NAME,NAME VALUE [,NAME VALUE]`

This sets Bus Interface parameter(s) as specified for all interfaces this port is part of. If the seconds variant is used, they are only be set for the names interface. Note that if it occurs, XIL_INTERFACENAME must be the first element in the list.

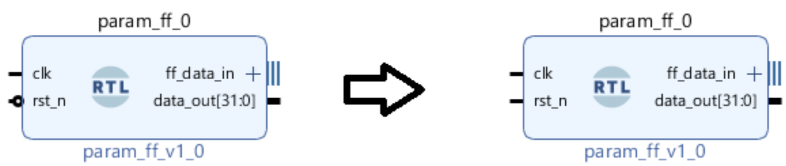
As an example let us assume that a reset port (`rst_n` in the code snippet below) has a polarity of active-Low and we want to override this polarity to active-High for all the interfaces that this `rst_n` port is a part of. This can be overridden as shown below. Note the setting on the attribute is called X_INTERFACE_PARAMETER.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity param_ff is
    generic(
            data_width : integer := 32);
    Port ( data_in : in STD_LOGIC_VECTOR ((data_width - 1) downto 0);
            clk : in STD_LOGIC;
            rst_n : in STD_LOGIC;
            data_out : out STD_LOGIC_VECTOR ((data_width - 1) downto 0));
end param_ff;

architecture Behavioral of param_ff is
    ATTRIBUTE X_INTERFACE_INFO : STRING;
    ATTRIBUTE X_INTERFACE_PARAMETER : STRING;
    ATTRIBUTE X_INTERFACE_INFO of data_in: SIGNAL is
"xilinx.com:user:ff_data_in:1.0
ff_data_in data_in";
    ATTRIBUTE X_INTERFACE_PARAMETER of rst_n: SIGNAL is "POLARITY
ACTIVE_HIGH";
begin
    process (rst_n, clk)
    begin
        if (rst_n = '0') then
            data_out <= (others => '0');
```

```
        elsif (rising_edge(clk)) then
            data_out <= data_in;
        end if;
    end process;
end Behavioral;
```

On the block design, the polarity of `rst_n`, which is inferred as active-Low by default, now changes to active-High (indicated by the bubble on the `rst_n` pin).

Figure 206: **Module Reference Highlighting X_INTERFACE_PARAMETER attribute**



X_INTERFACE_IGNORE

- Attach to: Port
- Syntax: `true|false`

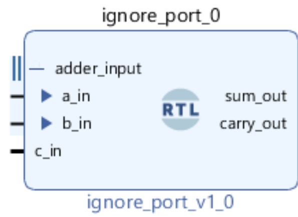
If set to `true`, this port will not be automatically added to any interface inferred by the heuristic.

In the following code snippet we have three input ports `a_in`, `b_in` and `c_in`, but we do not want to add the third port `c_in` to the interface.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ignore_port is
    Port ( a_in : in STD_LOGIC;
           b_in : in STD_LOGIC;
           c_in : in STD_LOGIC;
           sum_out : out STD_LOGIC;
           carry_out : out STD_LOGIC);
end ignore_port;

architecture Behavioral of ignore_port is
  ATTRIBUTE X_INTERFACE_INFO : STRING;
  ATTRIBUTE X_INTERFACE_IGNORE : STRING;
  ATTRIBUTE X_INTERFACE_INFO of a_in: SIGNAL is
"xilinx.com:user:adder_input:1.0
adder_input a_in";
  ATTRIBUTE X_INTERFACE_INFO of b_in: SIGNAL is
"xilinx.com:user:adder_input:1.0
adder_input b_in";
  ATTRIBUTE X_INTERFACE_IGNORE of c_in: SIGNAL is "true";
begin

  sum_out <= a_in OR b_in OR c_in;
  carry_out <= a_in XOR b_in XOR c_in;
end Behavioral;
```

When instantiated on the block design, this will be as shown below.

Figure 207: **Module Reference Highlighting X_INTERFACE_IGNORE attribute**



X_INTERFACE_MODE

- Attach to: Port
- Syntax: `MODE [MONITOR_MODE] [INTERFACE_NAME obsolete]`

The last parameter is obsolete and ignored.

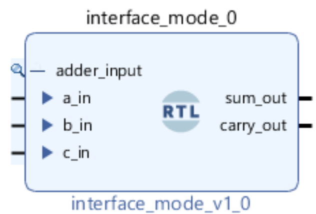
Sets the interface mode of all the interfaces that contain the port. Set only one MODE per interface; if there are more, they will be ignored altogether.

The following code snippet shows the usage.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

entity interface_mode is
    Port ( a_in : in STD_LOGIC;
           b_in : in STD_LOGIC;
           c_in : in STD_LOGIC;
           sum_out : out STD_LOGIC;
           carry_out : out STD_LOGIC);
end interface_mode;

architecture Behavioral of interface_mode is

  ATTRIBUTE X_INTERFACE_INFO : STRING;
  ATTRIBUTE X_INTERFACE_MODE : STRING;
  ATTRIBUTE X_INTERFACE_INFO of a_in: SIGNAL is
"xilinx.com:user:adder_input:1.0
adder_input a_in";
  ATTRIBUTE X_INTERFACE_INFO of b_in: SIGNAL is
"xilinx.com:user:adder_input:1.0
adder_input b_in";
  ATTRIBUTE X_INTERFACE_INFO of c_in: SIGNAL is
"xilinx.com:user:adder_input:1.0
adder_input c_in";
  ATTRIBUTE X_INTERFACE_MODE of c_in: SIGNAL is "monitor";

begin
  sum_out <= a_in OR b_in OR c_in;
  carry_out <= a_in XOR b_in XOR c_in;
end Behavioral;
```

The module reference module when instantiated on the block design, looks as follows. Note the magnifying glass icon on the cell to signify that the interface type if of "monitor".

Figure 208: **Module Reference Highlighting X_INTERFACE_MODE attribute**



X_INTERFACE_PRIORITY_LIST

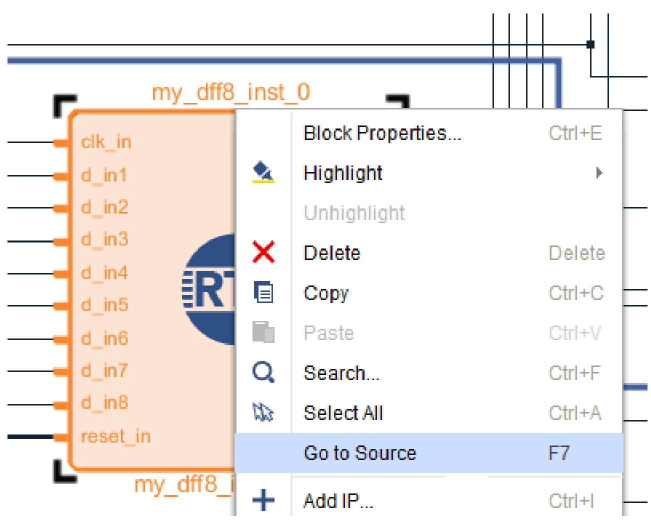
- Attach to: Component
- Syntax: `VLNV [VLNV VLNV etc]`

Specifies the priority order in which the heuristic tries to infer bus interfaces. The highest priority will be given to match the ports in the component first, in the order specified. This is the highest priority list and it overrides project settings, and repository order.

Editing the RTL Module After Instantiation

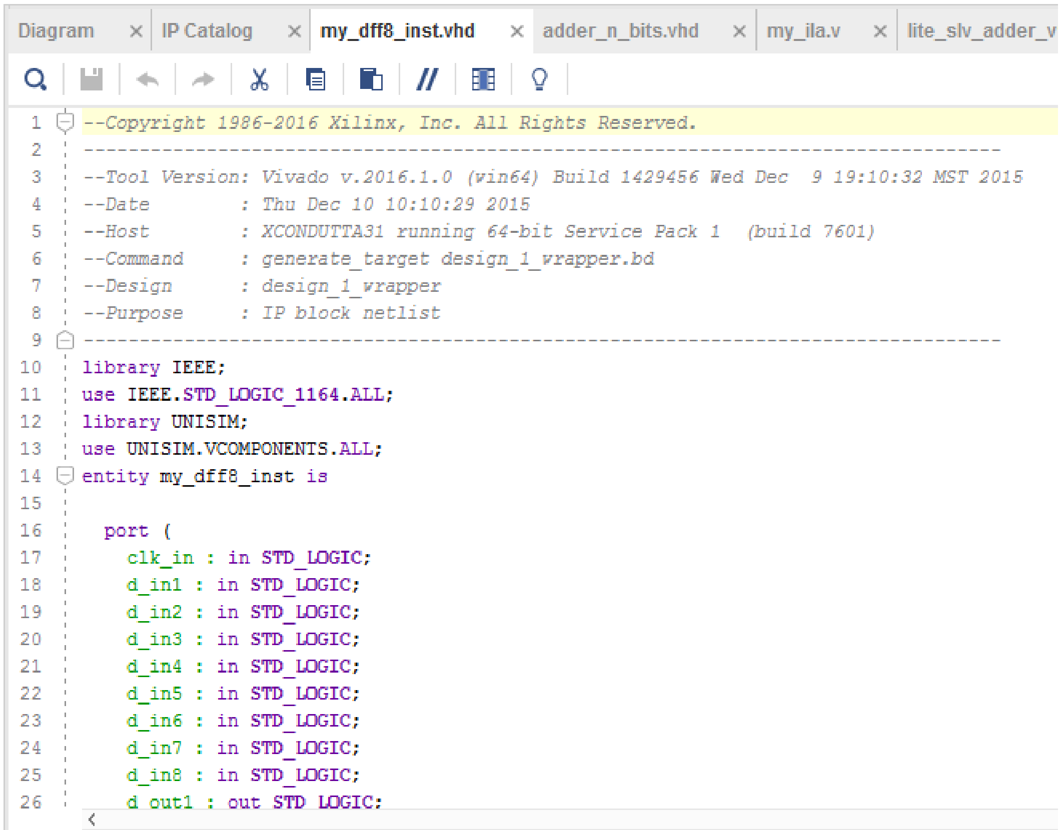
To edit the source code of a module, right-click it, and select **Go To Source** from the context menu, as shown in the following figure.

Figure 209: **Editing an RTL Module After Instantiation**



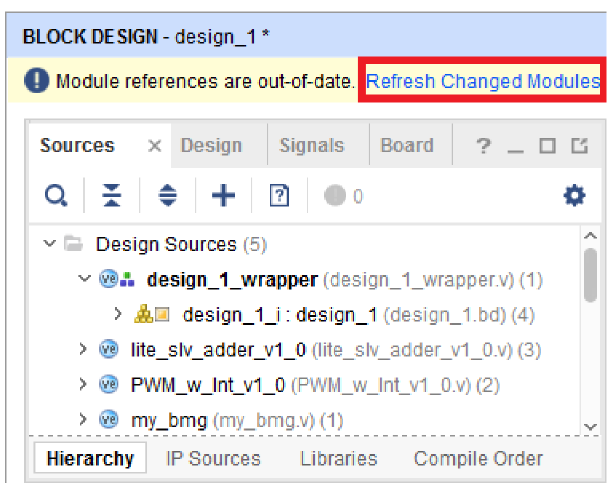
This opens the module source file for editing, shown in the following figure.

Figure 210: **Editing Top-Level Source File in the Editor**



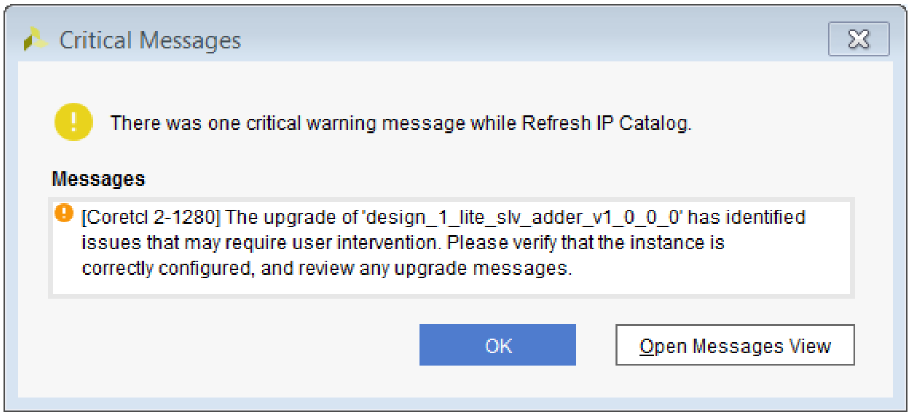
If you modify the source and save it, notice that the Refresh Changed Modules link becomes active in the banner of the block design canvas, as shown in the following figure.

Figure 211: **Updating an RTL Module**



Click **Refresh Changed Modules** to reread the module from the source file. Depending on the changes made to the module definition, for example, adding a new port to the module, you might see a message such as shown in the following figure.

Figure 212: **Critical Warning Dialog Box after Updating an RTL Module**



On the Tcl console, you see the changes that were made to the module, as shown in the following snippet:

```
WARNING: [IP_Flow 19-4698] Upgrade has added port 'new_port'
WARNING: [IP_Flow 19-3298] Detected external port differences while
upgrading
'module reference design_1_my_dff8_inst_0_0'. These changes may impact your
design.
CRITICAL WARNING: [Coretcl 2-1280] The upgrade of 'module reference
design_1_my_dff8_inst_0_0' has identified issues that may require user
intervention.
Please verify that the instance is correctly configured, and review any
upgrade
messages.
```

Module Reference in a Non-Project Flow

The following is a sample script for opening a block design that uses the Module Reference feature, and contains referenced modules.

IMPORTANT! *The RTL source files for the referenced modules must be read prior to opening the block design.*

```
# Specify part, language, board part (if using the board flow)
set_part xc7k325tffg900-2
set_property target_language VHDL [current_project]
set_property board_part xilinx.com:kc705:part0:0.9 [current_project]
set_property default_lib work [current_project]
```

```
# The following line is required for module reference and also for
# third-party synthesis flow
set_property source_mgmt_mode All [current_project]

# Read the RTL source files for referenced modules prior to reading
# and opening the Block Design
read_verilog *.v
read_vhdl *.vhdl

# Read and Open the Block Design
read_bd ./bd/mb_ex_1/mb_ex_1.bd
open_bd_design ./bd/mb_ex_1/mb_ex_1.bd

# Add the HDL Wrapper for the Block Design
read_vhdl ./bd/mb_ex_1/hdl/mb_ex_1_wrapper.vhd

# Write hardware definition
write_hwdef -file mb_ex_1_wrapper.hwdef
set_property source_mgmt_mode All [current_project]
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

# Implement
synth_design -top mb_ex_1_wrapper
opt_design
place_design
route_design
write_bitstream top

# For exporting the design to SDK, add the following commands.
write_mem_info ./top.mmi
file mkdir ./export_hw_np_mode/sdk
write_sysdef -hwdef mb_ex_1_wrapper.hwdef -bitfile top.bit -file
mb_ex_1_wrapper.hdf
```

Reusing a Block Design Containing a Module Reference

A block design that has RTL reference modules in it can be re-used in other projects, just like any other block design; however, you must first add the RTL module source files to the project, then add the block design to the project. This lets IP Integrator bind the cell instances present in the block design to the referenced RTL modules.

Handling Constraints in RTL Modules

Constraints are not automatically associated it with a module reference block. You need to add the appropriate constraints to the top-level project where the module reference block is instantiated. Associating a top-level XDC to a module reference requires that the file to be *scoped to the module*. By scoping, you are limiting the XDC to only work on the module reference.

RECOMMENDED: *Separate these constraints into another file. The scoping is by-reference or by-cell using the SCOPE_TO_REF or the SCOPE_TO_CELL property described in this link to "Appendix D, Editing or Overriding IP Sources" in the Vivado Design Suite User Guide: Designing with IP (UG896).*

All IP related constraints which are instantiated in a RTL block are automatically inferred and processed.

Limitations of the Module Reference Feature

The following limitations exist in the Module Reference feature:

- Because a module reference is not an IP, you cannot specify the Vendor, Library, Name and Version (VLNV).
- The RTL module definition cannot include netlists (EDIF or DCP), nested block designs (BD) or another module that is set as out-of-context (OOC) inside the RTL module.
- VHDL and Verilog are the only supported languages for module definition. A block design containing a module reference cannot be packaged as an IP. Instead, package the module as an IP separately, and then package the BD including that IP.
- Module Reference blocks cannot be opted out of upgrade while migrating a design from a previous version of Vivado.

TIP: *SystemVerilog and VHDL 2008 are not supported for the module or entity definition at the top-level of the RTL module.*

Chapter 14

Creating SDx Platforms Using Vivado

The SDx™ environment is an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems using Zynq-7000 SoCs and Zynq® UltraScale+™ MPSoCsZynq UltraScale™. The SDx IDE supports both the SDSoC (Software-Defined System On Chip) and SDAccel design flows on Linux and only SDSoC flows on Windows. The SDSoC system compiler (`sdscc` or `sds++`) generates an application-specific SoC by compiling application code written in C or C++ into hardware and software that extends a target platform.

Platforms target any application implemented in hardware using the SDx tools. Hardware components of a platform are designed using the Vivado Design Suite and IP Integrator. Software components of a platform are likewise created using the Software Development Kit (SDK) (SDK) or PetaLinux tool chain.

This chapter describes the flow to create hardware components of a platform using the IP Integrator. The design created using the IP Integrator captures the logical and physical interfaces to the hardware functions coming from the SDx environment. The processor, memory, and all external board interfaces are configured using a combination of Xilinx IP, custom IP, and RTL. This provides a logic âwrapper for the hardware functions to be executed properly on the platform. Many configuration and customization options exists on the types of hardware functions being accelerated.

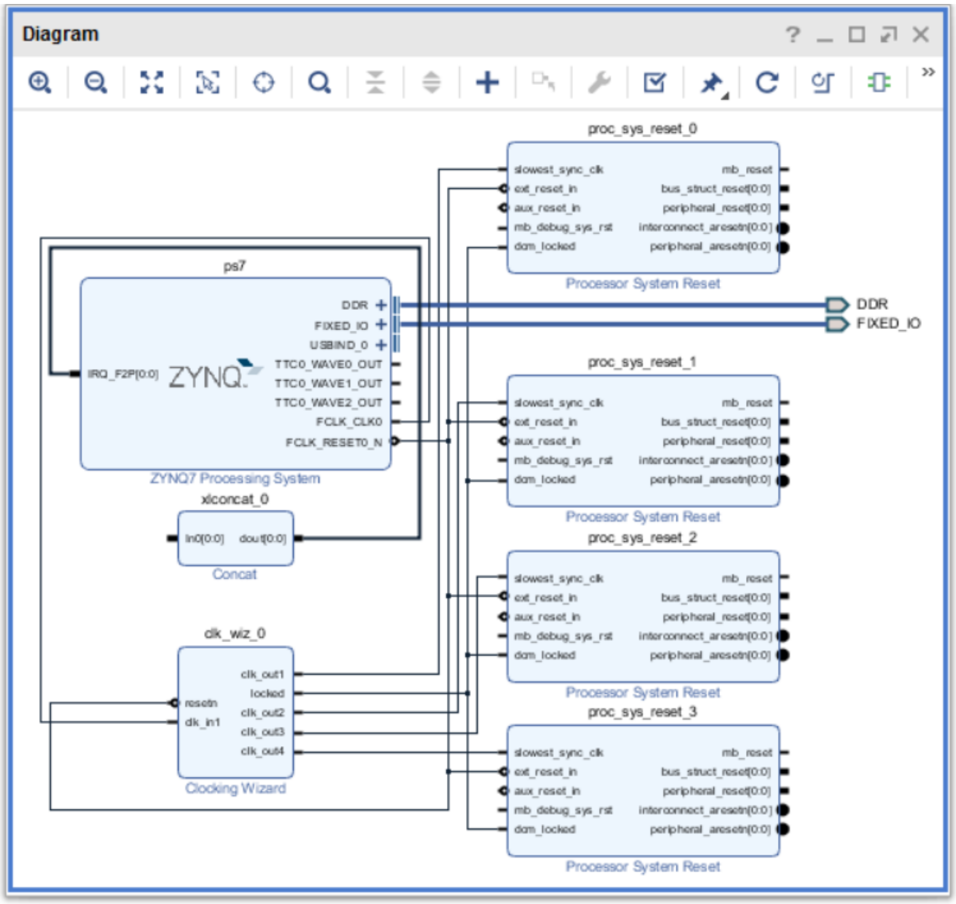
The Platform creation process is described in the *SDSoC Environment Platform Development Guide* (UG1146). This chapter covers the functionality available in Vivado to complete the hardware portion of the platform.

Creating the Vivado Platform Project

An SDSoC platform project begins with a Vivado Design Suite project file (`<platform>.xpr`) as the starting point to build the platform device support archive (DSA) file.

After the project is created, create a block design. The block design is used to instantiate the necessary IP to create the hardware portion of the platform. As an example, the following figure shows the block design for the base platform, ZC702, provided in the Vivado IP Integrator.

Figure 213: **IP Integrator Block Design of the ZC702 Platform**



There are a few things worth noticing in the block design above:

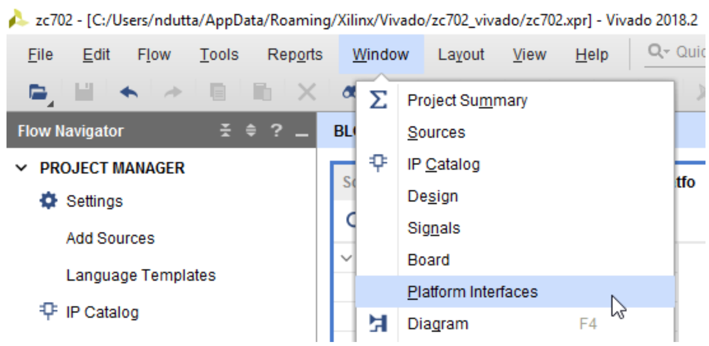
- The synchronized resets from the Processor System Reset blocks are not used anywhere in the design.
- Likewise, the input to the Concat block that is supposed to connect to interrupt sources are not connected.

These input and output pins are to be used by the hardware functions. If a hardware function uses a particular clock then it uses the synchronized reset output for that clock. After the hardware functions are built by the SDx tool, a final block design containing the hardware functions (packaged as an HLS IP) is instantiated in this block design, and all the necessary connections to clock, resets, interrupts and any AXI Interconnect needed are connected appropriately by the SDx build scripts.

Enabling the Platform Interfaces Window

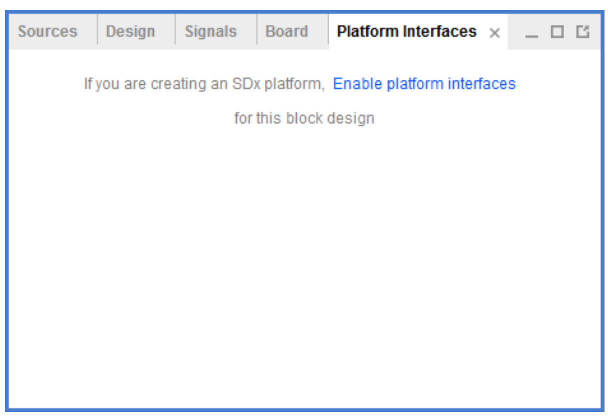
After the block design is complete, you can apply platform properties to different interfaces to be used by the hardware function(s) within the SDx environment. This can be done by enabling the Platform Interfaces tab. To enable the Platform Interface tab, select Window > Platform Interfaces from the menu, as shown in the following figure:

Figure 214: **Enabling the Platform Interface Tab**



This opens the Platform Interfaces tab, as shown in the following figure:

Figure 215: **Enabling the Platform Interface in the Platform Interface Tab**

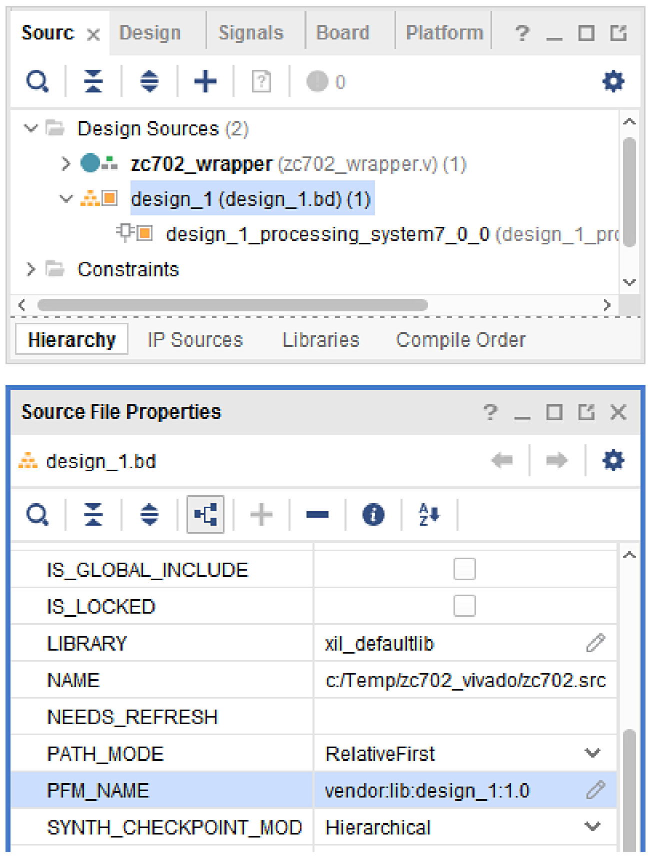


Click **Enable Platform Interfaces**. This enables the interfaces that are available within the block design and that can be enabled for use by SDx environment hardware accelerators. This action also applies the platform name property to the block design that represents the platform. The following set_property Tcl command is used to apply the platform name:

```
set_property pfm_name <bd_name> [get_files
{<path_to_project>\<project_name>.srcs\sources_1\bd\<bd_name>\<bd_name>.bd}]
```

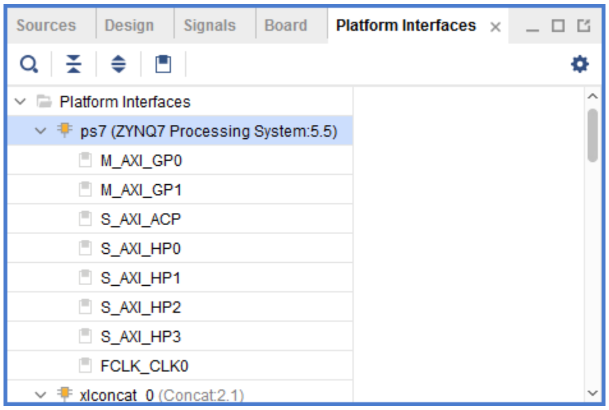
The Vendor, Library, Name, and Version of the platform can be changed by viewing the properties of the block design in the Source File Properties window. Select the block design in the Sources window and view the PFM_NAME property in the Source File Properties window. By clicking the pencil icon in the PFM_NAME field, you can edit and set the Vendor, Library, Name, and Version of the platform.

Figure 216: **Setting the Platform Vendor, Library, Name, And Version**



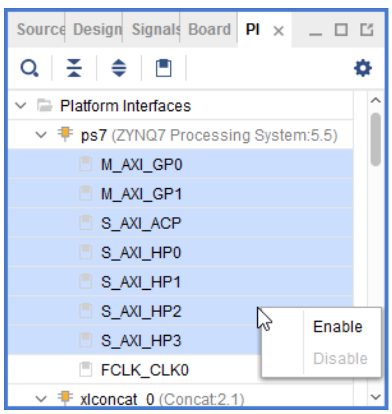
Setting Platform Properties on the PS7 Interfaces

Figure 217: **Platform Interfaces Tab with Available Interfaces**



To enable an interface, with the interface selected, right-click and select **Enable**.

Figure 218: **Enabling/Disabling interfaces available in the Platform**



You can select and enable multiple interfaces at the same time. As the interfaces are enabled, the Tcl Console reflects the Tcl commands used to apply the platform properties, as shown in the following code snippet:

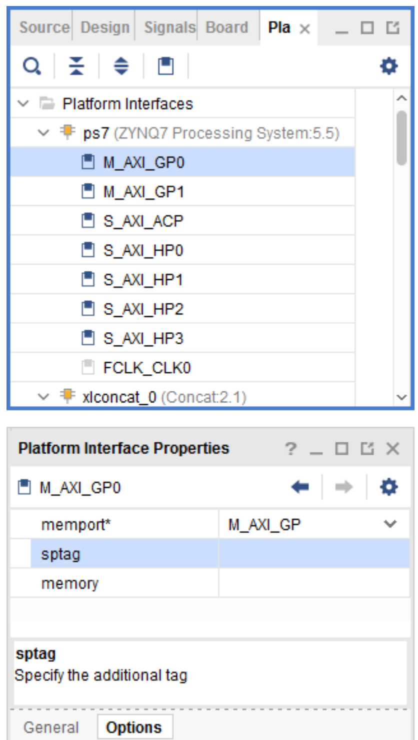
```
set_property PFM.AXI_PORT {M_AXI_GP0 {memport âM_AXI_GPâ sptag ââ memory
ââ} M_AXI_GP1
{memport âM_AXI_GPâ sptag ââ memory ââ} S_AXI_ACP {memport âS_AXI_ACPâ
sptag ââ memory ââ}
```

```
S_AXI_HP0 {memport âS_AXI_HPâ sptag ââ memory ââ} S_AXI_HP1 {memport
âS_AXI_HPâ sptag ââ
memory ââ} S_AXI_HP2 {memport âS_AXI_HPâ sptag ââ memory ââ} S_AXI_HP3
{memport âS_AXI_HPâ
sptag ââ memory ââ}} [get_bd_cells /ps7]
```

You can view and change properties through the Platform Interface Properties window. To view or change the properties, select the interface in the Platform Interfaces tab.

Note: Enabling an interface does not change the block design or the IP parameterization in any way. The block design captures this additional metadata so that the SDx tool knows what interfaces, clocks, resets, and so forth, are available to be used by the hardware functions.

Figure 219: **Modifying Properties for a Platform Interface**

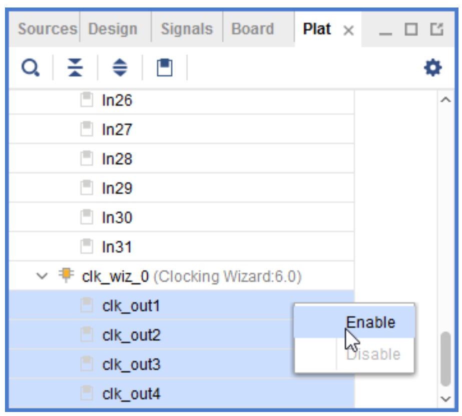


To disabled an enabled interface, right click the interface and select **Disable** from the context menu.

Setting Platform Properties on Clocks

A platform can have one or more clocks. Every clock must have a Processor System Reset IP block to synchronize the reset to these respective clock domains. As an example, in the following figure, there is a Clocking Wizard IP that has four clock outputs. The input frequency to the Clocking Wizard is the 50 MHz FCLK_CLK0 generated by the PS7. The Clocking Wizard generates four clocks with the following frequencies - clk_out1 (100 MHz), clk_out2 (142.857142 MHz), clk_out3 (166.666666 MHz), and clk_out4 (200 MHz). There is also one reset source generated by the FCLK_REST0_N in the PS7. This reset needs to be synchronized to all the four output clock sources in the design; therefore, four Processor System Reset blocks are used in the design. To make these clocks available to the SDx tool, select the desired clocks, right-click and select **Enable** in the Platform Interfaces tab, as shown in the following figure.

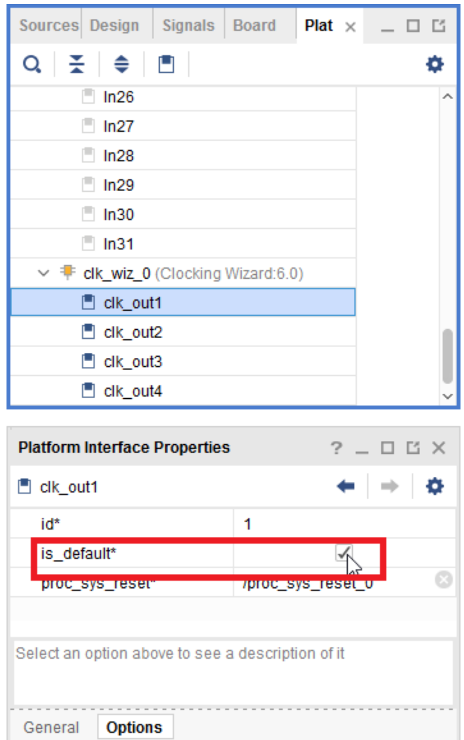
Figure 220: **Enabling Clocks in the Platform Interfaces Tab**




```
set_property PFM.CLOCK {clk_out1 {id â1â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_0â} clk_out2 {id â4â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_1â} clk_out3 {id â5â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_2â} clk_out4 {id â6â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_3â}} [get_bd_cells /clk_wiz_0]
```

As can be seen from the Tcl properties applied to these clocks, every one of these clocks has been assigned a id and the is_default property is set to false. In a platform only one clock must be a default clock. To assign the default clock in the platform, select the appropriate clock, and modify its is_default property from the Platform Interface Properties window by clicking the check-box, as shown in the following figure.

Figure 221: **Changing the is_default Parameter of a Clock**



The resulting Tcl command sets the is_default value of the desired clock to true, as shown in the following code snippet:

```
set_property PFM.CLOCK {clk_out1 {id â1â is_default âtrueâ proc_sys_reset
â/proc_sys_reset_0â} clk_out2 {id â4â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_1â} clk_out3 {id â5â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_2â} clk_out4 {id â6â is_default âfalseâ proc_sys_reset
â/proc_sys_reset_3â}} [get_bd_cells /clk_wiz_0]
```

If multiple clocks have the is_default parameter set to true, then during design validation you see the following error message:

```
ERROR: [BD 41-2087] Clocks '/clk_wiz_0/clk_out1, /clk_wiz_0/clk_out2, '
have the is_default
set to true. Only one clock can be set as a default clock in a platform
ERROR: [BD 41-2081] Cannot generate .hpfm file for current BD design(zc702)
ERROR: [Common 17-39] 'validate_bd_design' failed due to earlier errors.
```

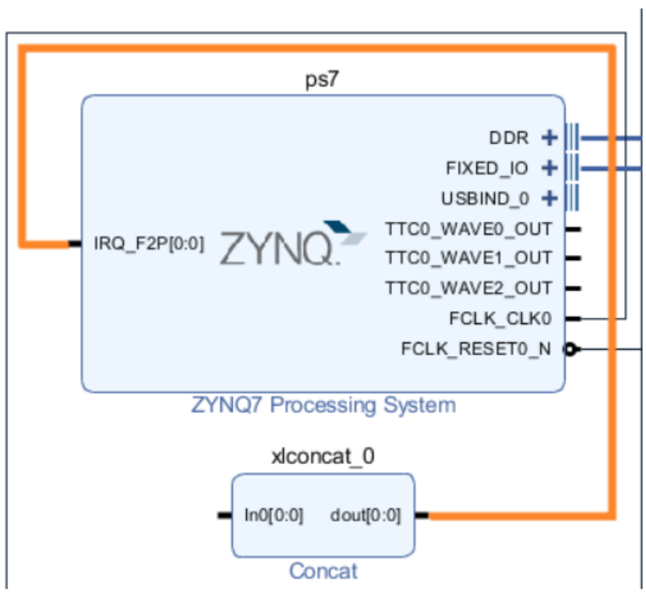
If none of the clocks have been set as a default clock then design validation fails with the following error message:

```
ERROR: [BD 41-2088] No default platform clock is selected. Please set
property is_default
to true for one of the platform clocks
ERROR: [BD 41-2081] Cannot generate .hpfm file for current BD design(zc702)
ERROR: [Common 17-39] 'validate_bd_design' failed due to earlier errors.
```

Setting Platform Properties on Interrupts

Interrupts in a ZynqÂ®-based design are funneled to the PS7 using a Concat block. A typical design looks as follows:

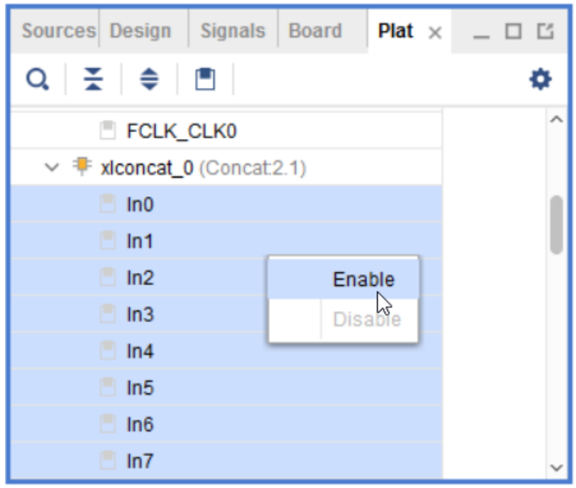
Figure 222: **Interrupts in a PS7 Based Design**



The input of the Concat block can connect to multiple interrupt sources. For a platform, you can leave the input of the Concat block unconnected such that interrupts from hardware functions can connect to this unconnected input.

To apply the platform properties on the Concat block, select the number of inputs, right-click, and select **Enable** in the Platform Interfaces tab, as shown in the following figure:

Figure 223: **Enabling Interrupts in a PS7 Based Design**



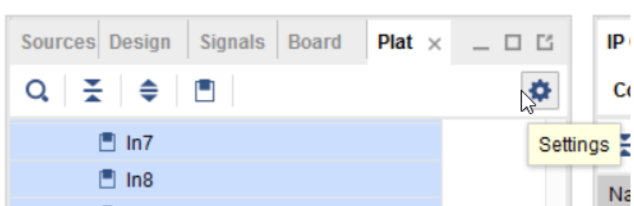
The resulting Tcl command shows the property applied to the Concat block inputs.

```
set_property PFM.IRQ {In0 {} In1 {} In2 {} In3 {} In4 {} In5 {} In6 {} In7
{} In8 {} In9 {}
In10 {} In11 {} In12 {} In13 {} In14 {} In15 {}} [get_bd_cells /xlconcat_0]
```

Filtering Interfaces in the Platform Interfaces Tab

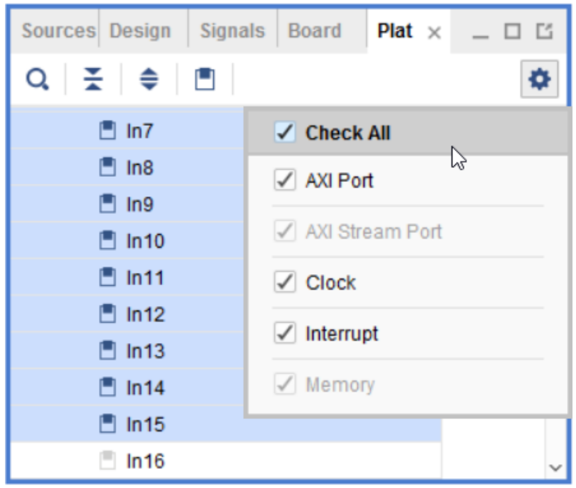
With multiple interfaces being available in the Platform Interfaces tab, the view may get a little cluttered. To filter the desired objects so that only they show up in the Platform Interfaces tab, click the Settings button in the top-right corner of the Platform Interface tab, as shown in the following figure:

Figure 224: **Filtering Interfaces using the Settings Button**



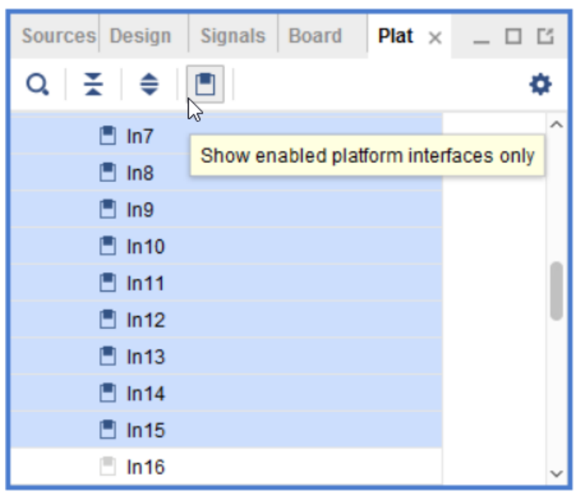
Clicking the Settings button brings up the menu to select or de-select the desired interfaces. The grayed out interfaces are interfaces that are not available in the particular platform.

Figure 225: **Filter by Un-Checking or Checking the Interfaces**



You can also choose to view only the objects that are enabled in the platform by clicking Show enabled platform interfaces only button.

Figure 226: **Showing Enabled Interfaces**



Validating the Block Design

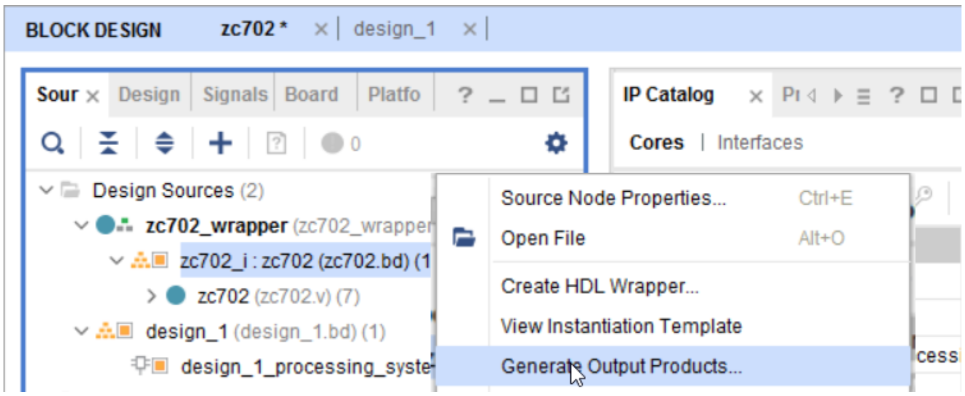
For a block design of a platform, use the `validate_bd_design -include_pfm` command to validate the block design by typing the command in the Tcl Console.

> **IMPORTANT!** *Do not validate the design using the tool bar for design validation, select* ***Tools → Validate Design****, or right-clicking in the block design canvas and from the context menu selecting Validate Design. These options do not use the required* `-include_pfm` *switch for design rules checks (DRCs) related to a platform.*

Generating the Block Design

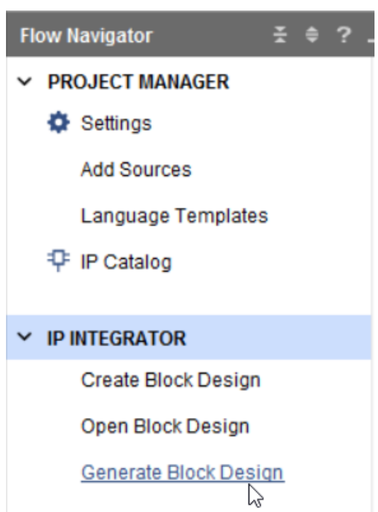
After the block design is validated, you can generate the block design. From the Sources window, select the block design, right-click, and select **Generate Output Products** from the context menu, as shown in the following figure:

Figure 227: **Generating Output Products**



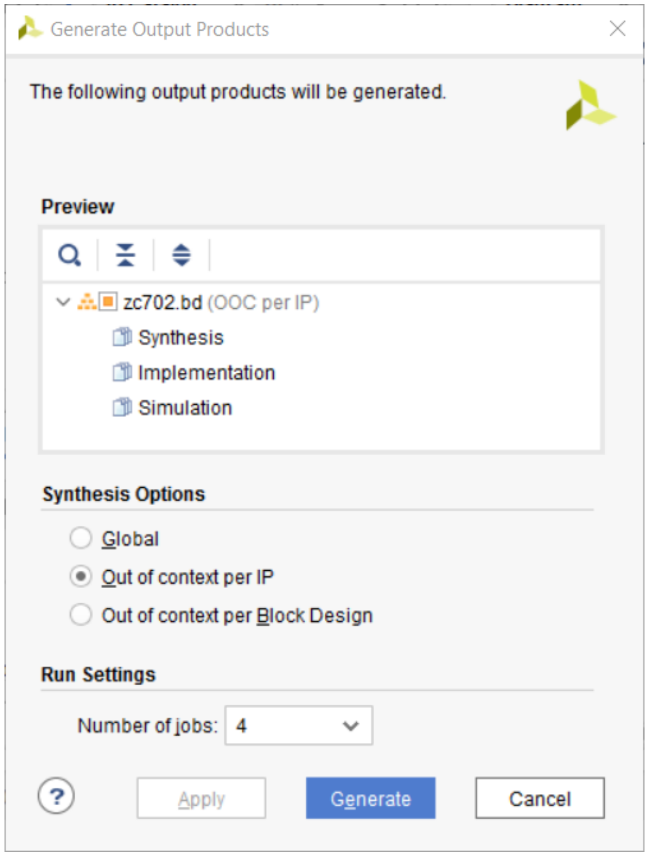
Alternatively, click **Flow Navigator → IP INTEGRATOR → Generate Block Design**, as shown.

Figure 228: **Generating Output Products from Flow Navigator**



The Generate Output Products dialog box opens from which you can select the synthesis option you want, and click **Generate**, as shown in the following figure:

Figure 229: **Generate Output Products Dialog Box**



After the output products are generated, you can create a top-level wrapper for the block design.

Writing the DSA

The device support archive (DSA) represents the hardware portion of the platform used in the SDx environment. The hardware platform design used to create a DSA consists of a Vivado IP Integrator subsystem design with all the required board interface IP cores configured and connected to the device I/Os.

You can write out the DSA after you implement the output product generation, or after the design has been implemented.

You can write the DSA by typing the `write_dsa` command at the Tcl Console, as shown:

```
write_dsa <path_to_dsa>/<dsa_name>.dsa
```

See the Vivado help for more information about this command.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
2. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
3. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
4. *Vivado Design Suite User Guide: Designing with IP* (UG896)
5. *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898)
6. *Vivado Design Suite User Guide: Using Constraints* (UG903)
7. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
8. *ISE to Vivado Design Suite Migration Guide* (UG911)
9. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
10. *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995)
11. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
12. *Generating Basic Software Platforms Reference Guide* (UG1138)
13. *Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions* (UG586)
14. *AXI Interrupt Controller (INTC) LogiCORE IP Product Guide* (PG099)
15. *UltraScale Architecture-Based FPGAs Memory IP LogiCORE IP Product Guide* (PG150)
16. *Integrated Logic Analyzer LogiCORE IP Product Guide* (PG172)
17. *System Integrated Logic Analyzer LogiCORE IP Product Guide* (PG261)
18. *LogiCORE IP Utility Vector Logic Product Brief* (PB046)
19. *LogiCORE IP Utility Reduced Logic Product Brief* (PB045)
20. *LogiCORE IP Constant Product Brief* (PB040)
21. *LogiCORE IP Concat Product Brief* (PB041)
22. *LogiCORE IP Slice Product Brief* (PB042)
23. *LogiCORE IP Utility Buffer Product Brief* (PB043)
24. *SDSoC Environment Platform Development Guide* (UG1146)
25. Vivado Design Suite Documentation

Training Resources

Xilinx® provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Vivado Design Suite QuickTake Video: Designing with Vivado® IP integrator
2. Vivado Design Suite QuickTake Video: Targeting Zynq® Devices Using Vivado®IP integrator
3. Vivado Design Suite QuickTake Video: AXI Interface Debug Using IP integrator
4. Designing FPGAs Using the Vivado Design Suite 2
5. Vivado Design Suite Embedded Systems Design Training Course
6. Vivado Design Suite Advanced Embedded Systems Design Training Course

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright